图灵程序设计丛书

游戏开发的
数学和物理

[日] 加藤洁 著　徐谦 译



告诉你，让游戏逼真、流畅的秘密

42个编程实例
打通游戏开发的任督二脉

134张图解
清晰讲解游戏编程五大基本实现

①物体的运动　②卷动　③碰撞检测
④光线的制作　⑤画面切换

人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：游戏开发的数学和物理

作者：［日］ 加藤洁

译者：徐谦

ISBN：978-7-115-37581-0

译者序

提到物理和数学，尤其是高等数学时，想必多数人的第一反应都是“好难”吧。译者本人也是如此，在看到如同天书一般的物理数学公式时，大都会望而却步，因为下意识里我们都会把纯理论性的物理数学问题，归纳为“科学家”才会研究的领域，从而不自觉地选择逃避。

仔细想想，物理和数学真的很难吗？绝大多数翻开这本书的朋友，都应该有中考、高考的经历，并且一路过关斩将应付过无数次物理和数学考试。从这个角度来看，其实我们每个人都已经具备了不错的物理和数学天赋，“好难”只是一个借口。

那么为什么我们会这样抗拒物理和数学呢？从很多朋友的反馈来看大致可以归纳为两点：没有用、没意思。比如大学的微积分课程，只会告诉我们如何求解积分习题，却几乎没有提到积分在现实世界中有什么实际用途；再比如我们在学校学习正态分布，只看到教科书上画着正态分布的图形，却不知道这样的图形能做什么。

而在本书中，一切都从实例出发，不讲任何空洞或者脱离现实的理论知识。读过本书后，就可以知道积分不仅可以用来做题，还可以精准地模拟出物体在重力作用下的运动轨迹；而原本枯燥的正态分布，在本书中却可以转换成火山喷发一样的酷炫效果。通过本书总计超过40个实例的讲解，不仅能学到必要的游戏开发知识，更重要的是还能让我们重新认识物理和数学这两门学科——物理和数学绝不只是应付考试的毫无用处的纯理论知识，而是解决很多实际问题时必不可少的工具。

作为一本游戏开发的入门书，本书也颇具特色。目前市面上的游戏开发入门书大致可分为两种：一种好像葵花宝典这样的速成武功，按照书中的讲解一步一步地操作，最后一般都能做出一个完整的小游戏。快则快矣，却不求甚解，很多算法原理以及细节都被一笔带过，虽然能出成果，却仍然知其然而不知其所以然，不容易进一步提高；另一种如同易筋经，偏理论而不重实践。即使整本书看完，可能还是不知道从何入手才能做出一个完整的游戏，入门时间太长。本书虽然在形式上偏向后者，但是仍然重视实战，所选择的案例如物体移动、卷动、碰撞检测、画面切换等都是游戏开发中最基础也最重要的组成部分，每个案例又有

一到两个变形或进阶实例，满足不同层次读者的需要。因此如果是以游戏开发为职业目标，想扎扎实实地打好基础的话，本书将会是一个不错的选择。

译者的本职工作侧重于 Web 开发，出于对游戏开发的兴趣完成了本书的翻译，如书中存在疏漏之处，还请读者不吝指出。

最后感谢在翻译过程中给予我支持和鼓励的妻子和猫咪，同时也感谢图灵公司各位编辑的共同努力，才让本书得以面世。希望所有对游戏开发感兴趣的读者都能从中获益。

徐谦

2014 年 10 月 14 日

前言

这是一本通过游戏开发实例，讲解数学与物理知识的书。数学与物理，两者都是游戏开发中不可或缺的。然而在实际的游戏制作中，并不能像在学校的物理数学课上那样，不管三七二十一先背下一堆可能用得着的公式或解题技巧。真正的游戏开发，总是先遇到一个需要实现的需求，然后再在寻求实现方法的过程中，去学习必要的数学和物理知识，即“先有需求，再掌握实现需求的工具”。对于普通人来说，想必这样的学习模式才是更加自然并有效的吧。

自古以来，数学都是伴随着一些颇具实用性的目的产生并发展起来的。例如在古代，建筑中必须要测算距离或长度，赋税或商业活动中必须计算款项、面积、体积等，数学正是在这些现实需求中一步步发展起来的。而物理学的初衷也是用算式这种客观的语言来描述身边发生的现象，同样起源于非常实际的需求。因此，从某种意义上来说，“先有需求，再去掌握实现需求的工具”这种学习模式，正是顺应整个数学和物理发展过程的最自然的方式。

时至今日，数学和物理已经经历过无数才华横溢的先人的反复锤炼，称它们为集合了人类智慧的瑰宝也不为过，它们共同支撑起了现代的科学技术，当然也是游戏以及计算机行业的基石。但是我们在学习这样的智慧瑰宝时，却往往不明白数学和物理究竟有什么作用，只是像背诵咒语一样强行记下公式，甚至也有不少人对其感到厌恶。其实人的本性确实如此，假如不是对某样事物抱有兴趣，或者迫于外部压力，是很难认真地自觉学习的。即便数学和物理是人类智慧的瑰宝，但如果在连它们能做什么都不了解的情况下去勉强学习，那也是违背人类本性的，学习起来自然也就倍感艰难了。

本书以编程为题材，每一章节都会先提出一个需要解决的实际问题，然后针对问题通过公式和必要的理论知识讲解解决方法。为了让这样的学习模式更加有效，书中所提出的问题应当是真正实用的，才容易让读者产生兴趣，因此本书所涉及的问题严格选取了实际 2D 游戏编程中不可缺少的技术。而本着尽量生动有趣的原则，在编撰人员的努力下，解决问题的示例程序都采用了与实际游戏同样的图片素材。希望广大读者朋友可以籍由本书更加自然地走进门槛不低的数学和物理世界。

本书大致可以有两种阅读方法。方法一，通过解决问题的实际案例入门（第 1 章～第 5 章），掌握必要的数学、物理知识。这种阅读方法适合所有想编写游戏程序的读者。针对这部分读者，本书涵盖了 2D 游戏开发中所涉及的几乎所有的必要知识。而对于想要开发 3D 游戏的读者，也建议不要一下跳跃到 3D，先以本书所涉及的 2D 知识为基础开始学习会更加容易入门。另一种阅读方法是，从本书的理论部分（第 6 章）开始，先了解本书所涉及的数学、物理公式在实际生活中究竟有什么作用。推荐那些在中学、大学里不得不学习理科，却又不知道学习理科有什么用的同学们采用这种阅读方法。我本人作为一名游戏学校的讲师，一直在帮助学生将中学、大学里的教育成果转变为游戏公司的实际生产力，充当着桥梁的作用，因此如果本书能对中学生、大学生的理科教育有一点帮助的话，对我来说将是意外的惊喜。

在阅读本书时请注意，为了简化理论知识以外的部分，本书中的示例代码可能会违背一些代码编写的基本规范。特别是类似将 vx 这样的短变量名作为全局变量的做法，在真正编程时千万不要去模仿。

最后，对给予我执笔机会，并对内容等方面提供巨大帮助的小川史晃编辑及翔泳社的诸位，对邀请我写作并协助策划的 Amusement Media 综合学院的猪狩贤一郎先生，以及对本书文字提出很多建议的学院剧本专业的老师们，一并表示深深的感谢。

2013 年 11 月

加藤洁

关于本书

目标读者·必要知识

本书面向的是那些想从事游戏开发，希望学习游戏开发中必要的数学和物理学知识的朋友。因此本书是一本从基础开始，讲解简单亲切的入门书。学习本书前，最好能掌握一些 C 语言的基础知识，如果读者不具备 C 语言知识，建议另外准备一些 C 语言（或者 C++）的资料，遇到不懂的部分可以边查资料边学习。

本书的构成

本书的 1～5 章会介绍游戏开发中的一些常见问题及解决方法。在介绍过程中会将 2D 游戏必需的知识一网打尽，同时还严格挑选出少量 3D 游戏编程的基础内容以供参考。读者可以通过调试示例程序，对照书中讲解一步步学习这些程序所用到的数学和物理学知识以及数学公式的用法。

本书的第 6 章则对所有示例中涉及的数学和物理学知识的基础理论进行一遍系统梳理，帮助读者更好地理解第 1～5 章。

正如前言中所说，本书有两种学习方法：方法一，先阅读 1～5 章，如果遇到不好理解的部分，再去第 6 章寻求更系统的说明。方法二，先阅读第 6 章对所有的基础理论有大致印象，然后在 1～5 章 中逐一印证这些基础理论是如何运用到实际案例中去的。读者朋友可以自行选择适合自己的阅读方式。

其他特色

阅读难度

本书每小节都有形象的难易度标识，在学习时请注意参考。



源代码

本书刊载的源代码，大多为了便于讲解进行了精简。不过书中的代码与实际的源代码文件行数编号是一一对应的，读者可以自行下载源代码对照学习。

下载地址：

https://github.com/AlloVince/physics_mathematics_skills_for_game_develop

在线查看：

https://github.com/AlloVince/physics_mathematics_skills_for_game_develop

另外，本书中的示例程序，除了数学和物理学知识外还使用了DirectX，这部分内容不在本书的范围内，请读者自行参考相关资料。

示例程序

本书每小节都会给出1～3个示例程序进行讲解，可能仅在纸上阅读代码理解起来有点困难，建议读者实际运行示例程序，并且修改源代码中的关键部分进行调试，对理解会更有帮助。

下载

示例程序均为可执行文件形式（EXE文件），同时附带源代码及图片素材，可以从上述网址进行下载。同时作为参考案例，源代码中还有一些额外的EXE文件，这些文件没有附带源码，请读者自行琢磨如何实现。

运行EXE文件需要安装DirectX 11。源代码的编 译、运行需要安装

DirectX 11 SDK。DirectX 11 的下载请参考下面的网址，编译、运行的说明请参考本书末尾的附录部分。

DirectX 11 下载地址：

http://www.microsoft.com/zh-cn/download/details.aspx?id=35

运行环境

运行示例程序前请确认已经安装有以下的运行环境。

OS：Windows 7(32/64 位) / Windows 8(64 位)

开发环境：

Visual Studio Express 2013 for Windows Desktop

Visual Studio Professional 2012 / Visual Studio 2010 Professional

DirectX SDK 9.29.1962 / DirectX 最终用户运行时 9.29.1974

免责声明

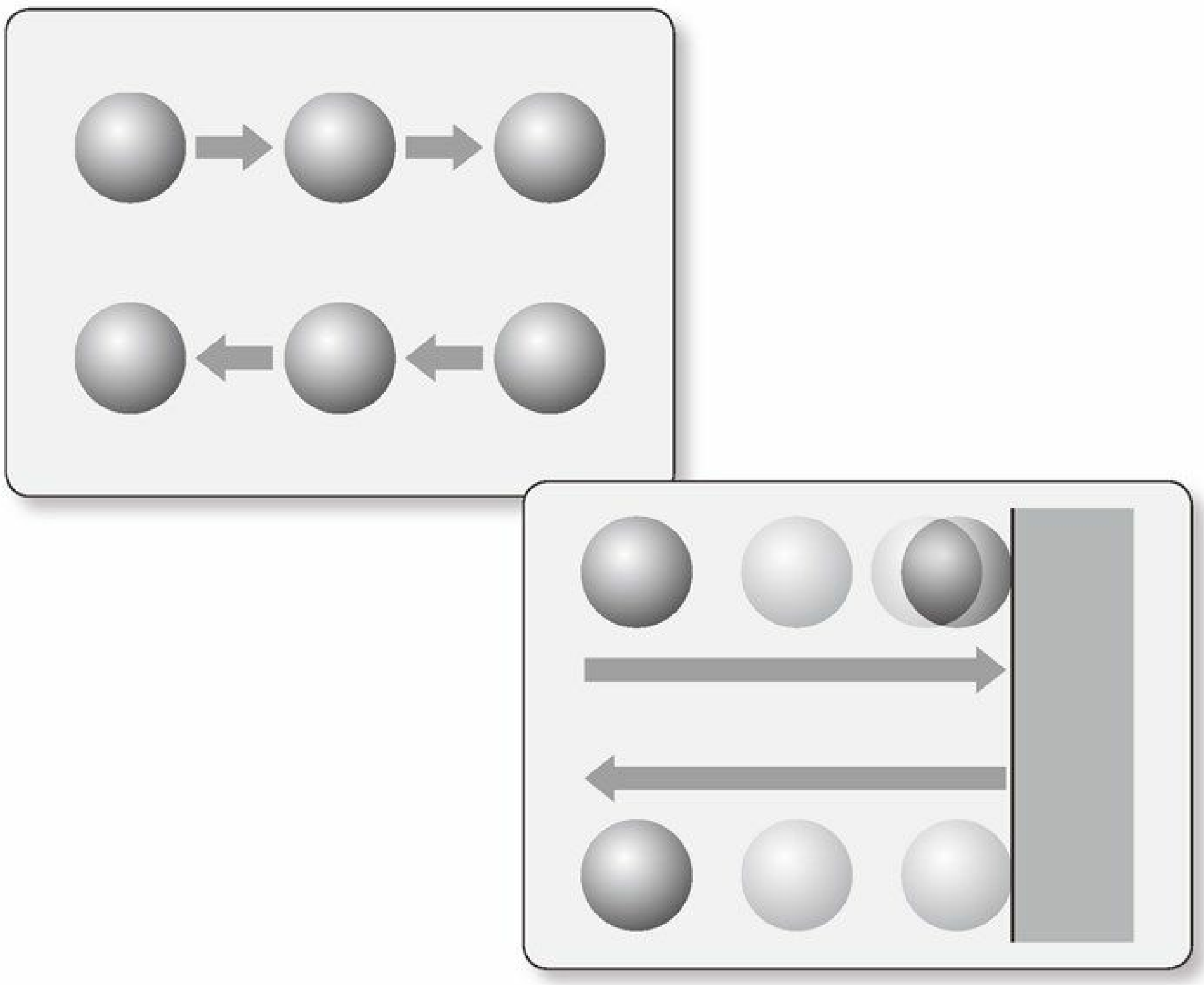
本书中的示例程序及源代码，已经获得出版社及作者的确认，读者可以作为普通用途使用。但是万一由于使用不当等产生损失，作译者、翔泳社和人民邮电出版社不承担相应责任。

关于著作权

第 **1** 章　物体的运动

1.1 让物体沿水平方向运动

Key Word 匀速直线运动、**x+=v;**、**v = -v** RANK/easy



物体运动中最基本的是直线运动。在本小节，我们就来一起学习**RPG**、射击、动作、解谜等所有游戏类型中最基本的匀速直线运动吧。

说到物体的基本运动，请试想一下物体以一定速度沿直线运动的情况。没错，这种物体以固定速度行进的直线运动，称为匀速直线运动 。本

小节就来讲解如何让物体进行匀速直线运动。



图 **1-1-1**　匀速直线运动的程序

- 沿水平方向运动的程序

  示例程序 Movement_1_1.cpp 是物体单纯地沿水平方向运动的程序。虽然代码有点长，但大部分都是为了操作 DirectX 以在画面上演示运动，真正决定物体运动的只有以下部分（代码清单 1-1-1）。

  代码清单 **1-1-1**　决定物体运动的处理（**Movement_1_1.cpp** 片段）

```
int InitCharacter( void )          // 只在程序开始时调用一次
{
    x = 0;                         // 物体的初始位置
    v = 3;                         // 物体在x方向的速度

    return 0;
```

```
}


int MoveCharacter( void )          // 每帧调用一次
{
    x += v;                        // 实际移动物体

    return 0;
}
```

下面对这部分代码详细说明一下。首先，InitCharacter 函数是一个只在程序初始化时被调用一次的函数，用于设定物体的初始位置及 *x* 方向的速度。初始位置

```
    x = 0;                         // 物体的初始位置
```

为 0 代表物体开始位于画面最左端。*x* 方向的速度

```
    v = 3;                         // 物体在x方向的速度
```

被设定为 3。

之后的 MoveCharacter 函数是一个在画面切换时，即每帧被调用的函数。这个函数进行的处理为

```
    x += v;                        // 实际移动物体
```

即向水平方向的位置 *x* 加入速度 *v* 。在这里，*v* 在初始设定中为 3，所以每次 MoveCharacter 被调用（即每帧）时 *x* 坐标都会增加 3。一般来说到下一个画面切换的时间即帧速率为 $\frac{1}{60}$ 秒，因此程序中物体会以每秒 180 像素的速度向右侧移动（参考图 1-1-2）。

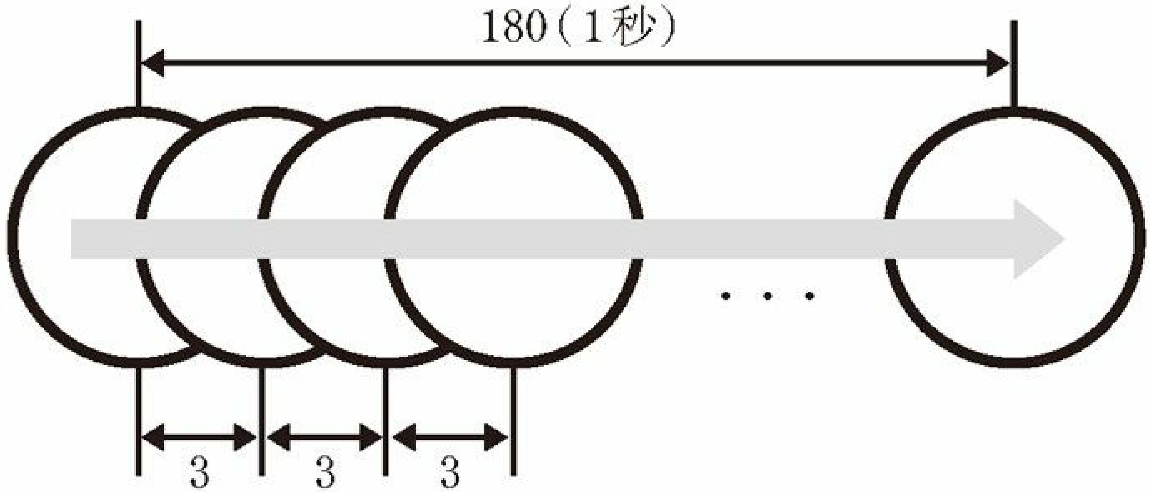


图 **1-1-2**　物体以每秒 **180** 像素的速度移动

- 实验程序

那么大家是不是对上述内容都理解了呢？来做个实验吧。首先让我们改变一下物体的运动速度。比如将 InitCharacter 函数内的

```
    v = 3;                              // 物体在x方向的速度
```

更改为：

```
    v = 1;                              // 物体在x方向的速度
```

这样一来物体的速度将变为原来的 $\frac{1}{3}$，即物体将比原来更加缓慢地移动（Movement_1_1a.cpp）。

然后让我们在不改变水平运动方式的前提下，尝试把物体动作改造得稍微复杂一点。原程序中，即使物体到达画面边缘，也不会停止，而是会直接移出画面。让我们来将其修改为：当物体碰到画面边缘，会沿反方向弹回。为此，我们分别对 InitCharacter 函数及 MoveCharacter 函数做以下改动。

代码清单 **1-1-2**　修改为物体碰到画面边缘时折回（**Movement_1_1b.cpp** 片段）

```
int InitCharacter( void )            // 只在程序开始时调用一次
{
    x = 0;                           // 物体的初始位置
    v = 10;                          // 物体在x方向的速度

    return 0;
}


int MoveCharacter( void )            // 每帧调用一次
{
    x += v;                          // 实际移动物体

    if ( x > VIEW_WIDTH - CHAR_WIDTH ) {          // 物体碰到右
        v = -v;                                   // 弹回
        x = VIEW_WIDTH - CHAR_WIDTH;              // 重设坐标为
    }
    if ( x < 0 ) {
        v = -v;
        x = 0;
    }

    return 0;
}
```

MoveCharacter 函数增加了一些内容，不过还是从 InitCharacter 函数开始按顺序说明。首先，InitCharacter 函数中 *x* 方向的速度

```
    v = 10;                              // 物体在x方向的速度
```

从之前的 3 增加到了 10。虽然速度变快了，但是由于新程序会让物体在画面边缘弹回，因此不必担心物体会一下子从画面中消失。然后在 MoveCharacter 函数中使用了 2 个 if 语句，追加了物体碰到画面左右端时弹回的处理。下面以画面右端的处理为例进行说明。

```
    if ( x > VIEW_WIDTH - CHAR_WIDTH ) {         // 物体碰到右端
```

这一行用来判断物体是否碰到画面右端。其中 VIEW_WIDTH 是画面的宽度，即画面右端的 *x* 坐标。那么有人可能会想，判断物体是否碰到画面右端，只要比较一下物体的 *x* 坐标和 VIEW_WIDTH 不就好了吗？事实上并没这么简单，因为电脑在绘制 2D 物体时，一般会以物体的左上角作为物体坐标的原点，如果只是使物体的 *x* 坐标不超出 VIEW_WIDTH，会让物体完全移出画面之外（参考图 1-1-3 左）。因此当物体碰到画面右端时，为了使程序做出“已经碰到”的判断，应当从画面右端的 *x* 坐标 VIEW_WIDTH 中，减去物体本身的大小 CHAR_WIDTH，然后再与物体的 *x* 坐标比较（参考图 1-1-3 右）。上面的讲解可能有点复杂，希望大家能正确理解。

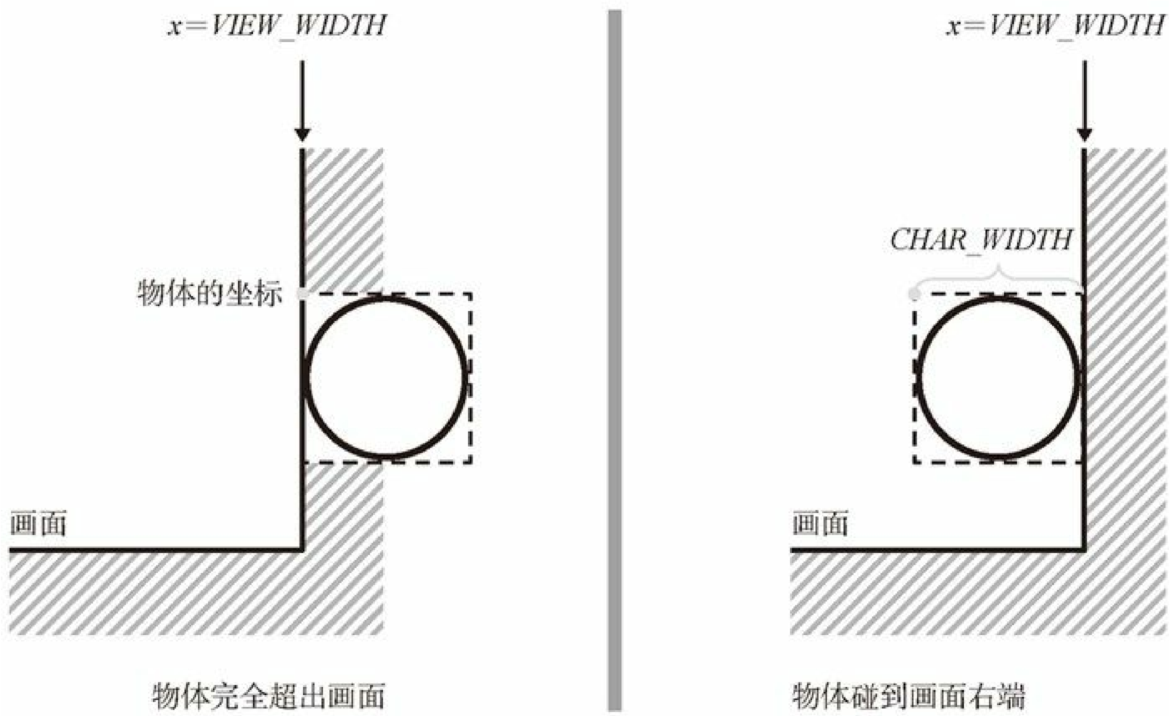


图 **1-1-3**　物体到达画面右端时的计算

然后，当物体碰到边缘时运行下面这行处理。

```
030  |              v = -v;                                    // 弹回
```

这行代码负责对物体的弹回动作进行处理。具体来说就是反转速度的符号。比如程序中速度的初始值为 10，当物体碰到画面边缘时速度将变成 -10，再碰到另一边时从 -10 变回 10，因此物体会沿反方向运动。可能有人会有疑问，既然这个 if 语句中判断的对象是画面右端，我们已经知道物体是向右运动的，同时物体在向右运动的过程中碰到边缘时的 *v* 必定为 10，将 *v* 变为 -10 后物体就会向左运动。那么将

```
030  |              v = -v;                                    // 弹回
```

直接写成

```
            v = -10;                              // 弹回
```

不是更加容易理解吗？这样的硬编码是不好的，因为如果这样写，我们下次要更改移动速度时，就不得不先在速度的初始化函数 InitCharacter 中找到下述设定项并进行更改，

```
    v = 10;                          // 物体在x方向的速度
```

但是只更改这里的初始设定程序仍然无法正常工作。比如，假设我们在此处将初始速度减半至 5，那么物体碰到画面右端的瞬间，其速度将突然倍增为 -10 并进行反向运动，这就违背了我们的意图。如果要扩展这个程序，比如新增加速度的处理等，这样硬编码的速度会让程序完全无法修改。因此考虑到程序的可维护性及扩展性，需要将速度书写为

```
            v = -v;                               // 弹回
```

最后说明下面这行。

```
            x = VIEW_WIDTH - CHAR_WIDTH;  // 重设坐标为画面边缘
```

正如注释中所写的那样，这行是为了将物体强制移动到正好与边缘接触的位置。之所以这样处理，是由于判断物体是否碰到边缘的 if 语句，是在物体实际已经碰到了边缘之后才执行的。比如，假设物体在前一帧中距离边缘还有 1 像素，且在当前帧中的速度达到了 10，那么当物体实际已经超出边缘 9 像素时，if 语句才会执行。这样显然会有问题。因此无论物体在与边缘接触的瞬间是否会超出，我们都将物体强制设置回与边缘正好接触的位置（参考图 1-1-3 右）。注意观察就会发现，重设物体位置的语句

```
            x = VIEW_WIDTH - CHAR_WIDTH;  // 重设坐标为画面边缘
```

与检测物体是否碰到右端的 if 语句

```
    if ( x > VIEW_WIDTH - CHAR_WIDTH ) {          // 物体碰到右端
```

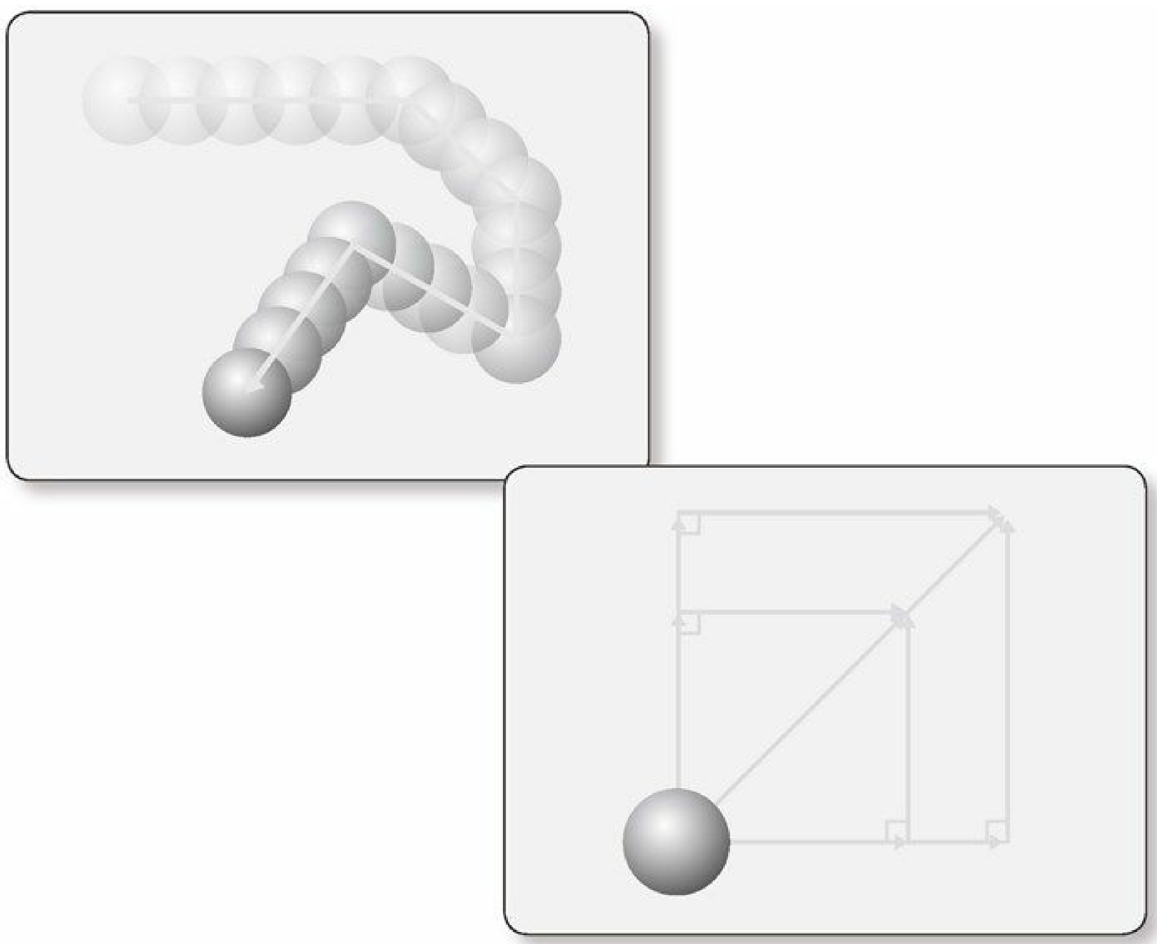
的区别只是将语句中的大于号（>）更改为了等号（=）。换个角度思考，这里的处理就相当于一旦做出物体接触到边缘的判断，就立即将物体强制移动至判断开始发生的位置。

至此，我们对物体碰到画面右端时弹回的处理进行了说明。这部分处理之后，代码中还有画面左端的弹回处理，处理内容与画面右端是一样的，此处不再赘述。

• 1.2　通过键盘控制物体的运动

Key Word　键盘输入、斜方向移动、勾股定理

RANK/easy 前半部分　RANK/normal 后半部分



物体运动中最基本的是直线运动。在本小节，我们就来一起学习**RPG**、射击、动作、解谜等所有游戏类型中最基本的匀速直线运动吧。

用户通过键盘输入控制物体的运动，是无法简单地通过直线运动实现的。本小节就来讲解包括斜方向运动在内的可通过键盘控制的物

体运动。

通过用户输入来控制物体运动是所有游戏的基础，从实用性来讲是极其重要的。

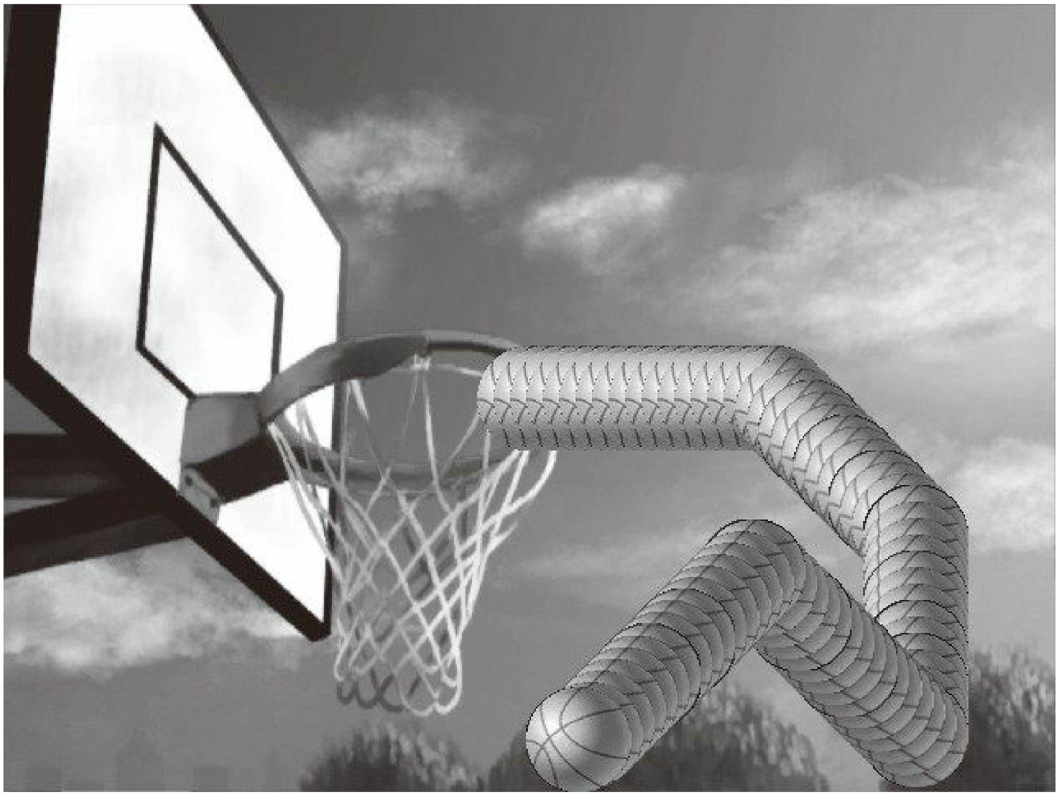


图 **1-2-1**　通过键盘输入控制物体运动的程序

- 通过键盘输入控制物体运动的程序

  示例程序 Movement_2_1.cpp 是一个通过键盘输入控制物体运动的简单程序。这个程序有点类似于一个极为简陋的射击游戏，按键盘的左右方向键可以移动物体。

  决定物体移动的代码如下所示（代码清单 1-2-1）。

  代码清单 **1-2-1**　根据键盘输入左右移动物体的处理（**Movement_2_1.cpp** 片段）

```
int InitCharacter( void )           // 只在程序开始时调用一
{
```

```
    // 物体的初始位置
    x = ( VIEW_WIDTH - CHAR_WIDTH ) / 2.0f;
    y = ( VIEW_HEIGHT - CHAR_HEIGHT ) / 2.0f;

    return 0;
}


int MoveCharacter( void )          // 每帧调用一次
{
    // 左方向键被按下时向左移动
    if ( GetAsyncKeyState( VK_LEFT ) ) {
            x -= PLAYER_VEL;
            if ( x < 0.0f ) {
                x = 0.0f;
            }
    }
    // 右方向键被按下时向右移动
    if ( GetAsyncKeyState( VK_RIGHT ) ) {
            x += PLAYER_VEL;
            if ( x > ( float )( VIEW_WIDTH - CHAR_WIDTH )
                x = ( float )( VIEW_WIDTH - CHAR_WIDTH );
            }
    }

    return 0;
}
```

在初始化函数 InitCharacter 中，定义了物体的初始位置，如下所示。

```
    // 物体的初始位置
    x = ( VIEW_WIDTH - CHAR_WIDTH ) / 2.0f;
    y = ( VIEW_HEIGHT - CHAR_HEIGHT ) / 2.0f;
```

即同时指定物体在水平方向和垂直方向的值，让物体的初始位置位于画面的中央。

然后在每帧调用的 MoveCharacter 函数中实现了一个主要功能：当一些特殊按键被按下时，物体向特定的方向移动；而除此以外的按键被按下时，则不做任何处理。因此必须要有一个按键检测机制来判断当前特殊按键是否被按下了。为此程序中调用了一个名为 GetAsyncKeyState 的函数。这个函数属于 Windows API 的一部分，即使没有 DirectX 也可以使用。如果想通过 DirectX 检测键盘输入等用户输入，需要使用 DirectInput 函数，而此处要检测的输入比较简单，仅使用 Windows API 就足够了。通过 GetAsyncKeyState 函数检测左方向键是否被按下，可以参考下面这行代码。

```
    if ( GetAsyncKeyState( VK_LEFT ) ) {
```

同理，检测右方向键时的代码为

```
    if ( GetAsyncKeyState( VK_RIGHT ) ) {
```

接着看 MoveCharacter 的内部，当左方向键被按下时，

```
        x -= PLAYER_VEL;
```

物体的 *x* 坐标会减去常数 PLAYER_VEL。对 *x* 坐标做减法就

等同于物体向画面的左方移动。但除此之外，还会进行下面这一处理。

```
if ( x < 0.0f ) {
    x = 0.0f;
}
```

这个处理会使物体到达画面左端时不再向左移动。

右方向键被按下（即满足条件 if (GetAsyncKeyState(VK_RIGHT))）时的处理为

```
x += PLAYER_VEL;
```

物体的 *x* 坐标将增加 PLAYER_VEL，物体会向右移动。此处也存在一个特殊处理，即

```
if ( x > ( float )( VIEW_WIDTH - CHAR_WIDTH )
    x = ( float )( VIEW_WIDTH - CHAR_WIDTH );
}
```

这样一来，物体在到达画面右端时，也不会再向右移动。另外，此处还进行了一个 (float) 强制类型转换，这是由于表示坐标的变量 *x* 使用的是 float 类型，而 VIEW_WIDTH 和 CHAR_WIDTH 则被定义为了 int 型的常数，为了比较和赋值，需要提前统一为 float 型。

- 多个按键的输入

  接下来让我们尝试使物体不只可以左右方向移动，还可以上下方向移动（Movement_2_1a.cpp）。为此，对 MoveCharacter 函数做如下变更（代码清单 1-2-2）。

  代码清单 **1-2-2**　使物体可以根据按键输入上下方向运动（**Movement_2_1a.cpp** 片段）

```
int MoveCharacter( void )         // 每帧调用一次
{
    // 左方向键被按下时向左移动------------------------------
    if ( GetAsyncKeyState( VK_LEFT ) ) {
            x -= PLAYER_VEL;
            if ( x < 0.0f ) {
                x = 0.0f;
            }
    }-------------------------------------------------------
    // 右方向键被按下时向右移动------------------------------
    if ( GetAsyncKeyState( VK_RIGHT ) ) {
            x += PLAYER_VEL;
            if ( x > ( float )( VIEW_WIDTH - CHAR_WIDTH )
                x = ( float )( VIEW_WIDTH - CHAR_WIDTH );
            }
    }-------------------------------------------------------
    // 上方向键被按下时向上移动------------------------------
    if ( GetAsyncKeyState( VK_UP ) ) {
            y -= PLAYER_VEL;
            if ( y < 0.0f ){
                y = 0.0f;
            }
    }-------------------------------------------------------
   // 下方向键被按下时向下移动-------------------------------
    if ( GetAsyncKeyState( VK_DOWN ) ) {
            y += PLAYER_VEL;
            if ( y > ( float )( VIEW_HEIGHT  - CHAR_HEIGHT
                y = ( float )( VIEW_HEIGHT - CHAR_HEIGHT )
            }
    }-------------------------------------------------------

    return 0;
}
```

代码行数增加了不少，不过基本上只是把修改前对 *x* 坐标的处理照搬到了 *y* 坐标上而已。具体说来代码段③的功能是，在上方向键被按下时从 *y* 坐标减去 PLAYER_VEL，并使物体在 到达画面上端后不再向上移动。④的部分同理，在下方向键被按下时向 *y* 坐标增加 PLAYER_VEL，并使物体在到达画面下端后不再向下移动。

这样一来很多人都会产生疑问，如果同时按下多个按键物体将会如何移动呢？比如物体只在水平方向移动时，同时按下左右方向键（可以看作停止运动的操作），物体会停止移动，这也比较符合人们一般的思维习惯。而如果物体不仅左右移动而且上下移动，那么就可以将左右按键与上下按键进行一些组合，可能性也就更多了，比如同时按上方向键与左方向键时物体会如何运动呢？在目前的程序里，如果同时按上方向键与左方向键，即代码中的条件①与条件③同时满足时，物体将向画面的左上角移动（参考图 1-2-2）。

想必物体的上述运动大家都能够想象得出来，但问题是物体的速度会如何变化呢？比如上方向键与左方向键同时被按下时，

```
            x -= PLAYER_VEL;
```

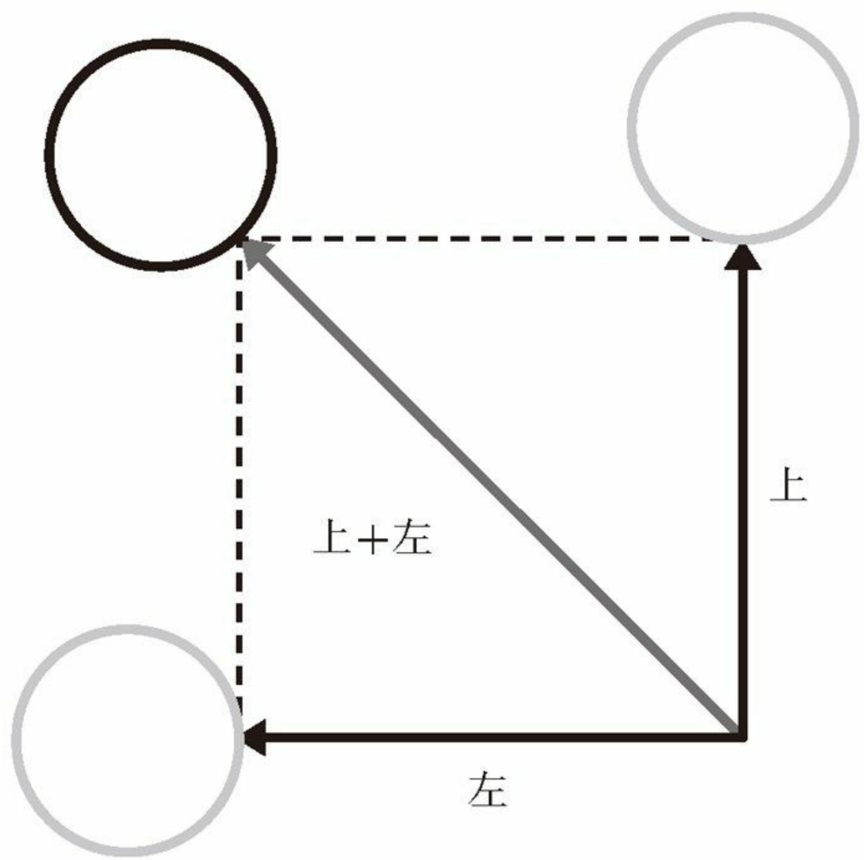


图 **1-2-2**　上方向键与左方向键同时被按下时物体向左上角移动

与

```
045  |           y -= PLAYER_VEL;
```

的处理会同时进行，那么物体的速度显然要比单方向的 PLAYER_VEL 更快，因为物体既在 *x* 方向上以 PLAYER_VEL 的速度运动，同时又在 *y* 方向上以 PLAYER_VEL 的速度运动。这种情况下，物体沿斜方向移动的速度究竟是多少呢？

首先可以明确的是，上述情况下物体的速度肯定要比 PLAYER_VEL 更快。但是由于物体不是在同一方向（如 *x* 方

向）上以 2 倍的 PLAYER_VEL 速度运动，而是在 x 方向和 y 方向上 分别以 PLAYER_VEL 的速度运动，所以恐怕物体的真正速度又比 2 倍的 PLAYER_VEL 要慢。于是可以得到下面的不等式。

$v_p < v < 2v_p$

这里 v_p 代表 PLAYER_VEL，即只按一个方向键时物体的速度。v 则代表两个方向键同时被按下时物体的速度。

但是，仅有这个不等式，还是无法知道斜方向上的速度 v 究竟是多少，或者说 v 与 v_p 之间究竟有怎样的关系。这个时候，我们就需要使用数学上的勾股定理 了。简单地说，勾股定理就是能通过直角三角形的两条边的长度，计算出剩下一条边的长度的定理。而这个定理在游戏中，以计算直角三角形的斜边的长度居多。假设直角三角形的斜边长为 c ，两直角边长分别为 a 、b ，那么公式为

$c^2 = a^2 + b^2$

参考图 1-2-3。

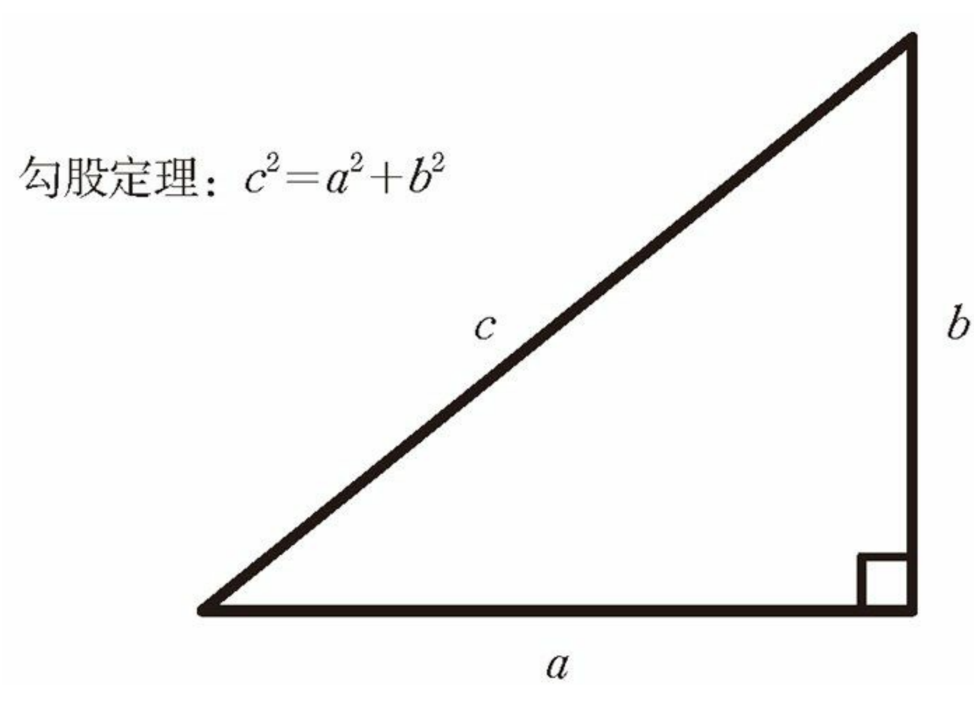


图 **1-2-3**　通过两边的长度求剩余一边的长度

而物体同时在 *x* 方向和 *y* 方向上以速度 v_p 运动的情况下，也可以套用勾股定理。首先，让我们只考虑一帧内物体运动的距离。此时，由于物体在一帧内在 *x* 方向与 *y* 方向上移动的速度均为 v_p（PLAYER_VEL），那么以这两个移动距离各自作为三角形的一边，就可以绘制出一个等腰三角形。而由于 *x* 方向与 *y* 方向互为直角，结果物体的最终速度 *v* 就是这个等腰直角三角形的斜边（参考图 1-2-4）。

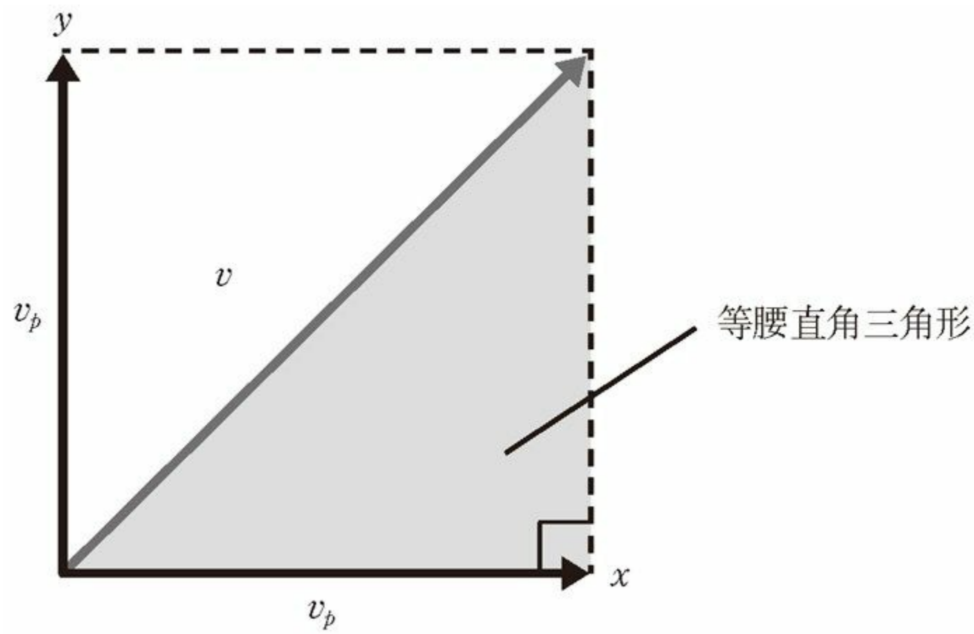


图 **1-2-4**　通过 ***x*** 方向与 ***y*** 方向的速度求速度 ***v***

套用勾股定理就是

$$v^2 = v_p^2 + v_p^2$$

等式右边相加得到

$$v^2 = 2v_p^2$$

因为我们最终要得到的是 *v* 而不是 v^2，所以等式两边都取平方根，得到

$$v = \pm\sqrt{2}v_p$$

由于我们最终要求的速度不会是负值，所以

$$v = \sqrt{2}v_p$$

即同时按下左方向键和上方向键时，物体既不会向水平方向移动，也不会向垂直方向移动，而是会以 $\sqrt{2}v_p$ 的速度向左上方向移动。$\sqrt{2}$ 约等于 1.41421，因此物体在斜方向的运动速度约为平时的 1.4 倍。在真正的游戏中（特别是 2D 射击类游戏），“斜方向的移动速度为原速度的 1.4 倍”这种情况一般是允许出现的（视实际情况也会有特例），大家随便想想就能列举出不少这种沿斜方向移动时速度变快的游戏。虽然以 $\sqrt{2}$ 倍的速度沿斜方向移动没有什么问题，但既然做到了这种程度，我们就来尝试一下如何让斜方向的移动速度保持不变吧。如 Movement_2_1b.cpp 所示，我们对 MoveCharacter 函数做了以下修改（代码清单 1-2-3）。

代码清单 **1-2-3**　使斜方向的移动速度保持不变（**Movement_2_1b.cpp** 片段）

```
#define ROOT2           1.41421f
int MoveCharacter( void ) // 每帧调用一次
{
    short        bLeftKey, bRightKey;
    short        bUpKey, bDownKey;

    bLeftKey = GetAsyncKeyState( VK_LEFT );
    bRightKey = GetAsyncKeyState( VK_RIGHT );
    bUpKey = GetAsyncKeyState( VK_UP );
    bDownKey = GetAsyncKeyState( VK_DOWN );
    // 左方向键被按下时向左移动-----------------------------
    if ( bLeftKey ) {
        if ( bUpKey || bDownKey ) {
                x -= PLAYER_VEL / ROOT2;
        }
        else {
                x -= PLAYER_VEL;
        }
                if ( x < 0 ) {
                    x = 0;
                }
```

```
    }--------------------------------------------------------
    // 右方向键被按下时向右移动--------------------------------
    if ( bRightKey ) {
        if ( bUpKey || bDownKey ) {
                x += PLAYER_VEL / ROOT2;
        }
        else {
                x += PLAYER_VEL;
        }
        if ( x >= ( float )( VIEW_WIDTH - CHAR_WIDTH ) ) {
                x = ( float )( VIEW_WIDTH - CHAR_WIDTH );
        }
    }--------------------------------------------------------
    // 上方向键被按下时向上移动--------------------------------
    if ( bUpKey ) {
       if ( bLeftKey || bRightKey ) {
                y -= PLAYER_VEL / ROOT2;
        }
        else {
                y -= PLAYER_VEL;
        }
        if ( y < 0 ) {
                y = 0;
        }
    }--------------------------------------------------------
    // 下方向键被按下时向下移动--------------------------------
    if ( bDownKey ) {
        if ( bLeftKey || bRightKey ) {
                y += PLAYER_VEL / ROOT2;
        }
        else {
                y += PLAYER_VEL;
        }
        if ( y >= ( float )( VIEW_HEIGHT - CHAR_HEIGHT ) )
                y = ( float )( VIEW_HEIGHT - CHAR_HEIGHT )
        }
    }--------------------------------------------------------

    return 0;
}
```

这里首先定义了 bLeftKey、bRightKey、bUpKey、bDownKey

四个变量，分别存放左、右、上、下方向键当前是否被按下这一信息。下面主要以左方向键被按下时向左移动的代码段①为例来详细说明。在这部分中，首先检查左方向键有没有被按下。

```
    if ( bLeftKey ) {
```

当左方向键被按下时，再接着检查上下方向键有没有被按下。

```
        if ( bUpKey || bDownKey ) {
```

如果在左方向键被按下的同时，又有上下方向键中的一个被按下，那么程序就会认为物体在向斜方向移动并执行以下语句。

```
                x -= PLAYER_VEL / ROOT2;
```

如果物体在 x 方向和 y 方向的移动速度仍然是 PLAYER_VEL，那么斜方向的运动速度就为原来的 $\sqrt{2}$ 倍，因此这里通过将原速度乘以 $\frac{1}{\sqrt{2}}$，斜方向的运动速度就会仍然保持 PLAYER_VEL 不变（参考图 1-2-5）。

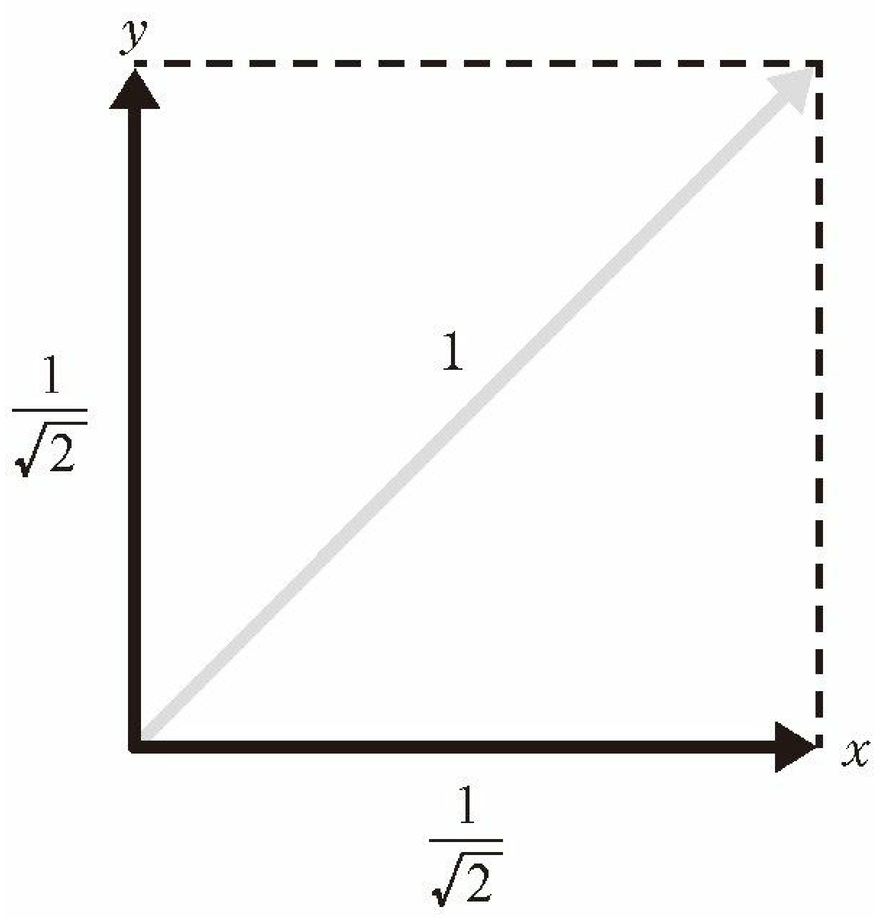


图 **1-2-5**　重置物体在水平及垂直方向的移动速度为 $\frac{1}{\sqrt{2}}$，使物体在斜方向上的移动速度保持不变

而如果左方向键被按下，上下方向键都没有被按下时，物体将仍然以速度 PLAYER_VEL 向水平方向运动。

```
                    x -= PLAYER_VEL;
```

通过将上述处理应用到右方向键、上方向键和下方向键，最终就可以让物体在水平、垂直、斜方向上的移动速度都固定为 PLAYER_VEL。但是上面的程序只能应对两个方向键被同时

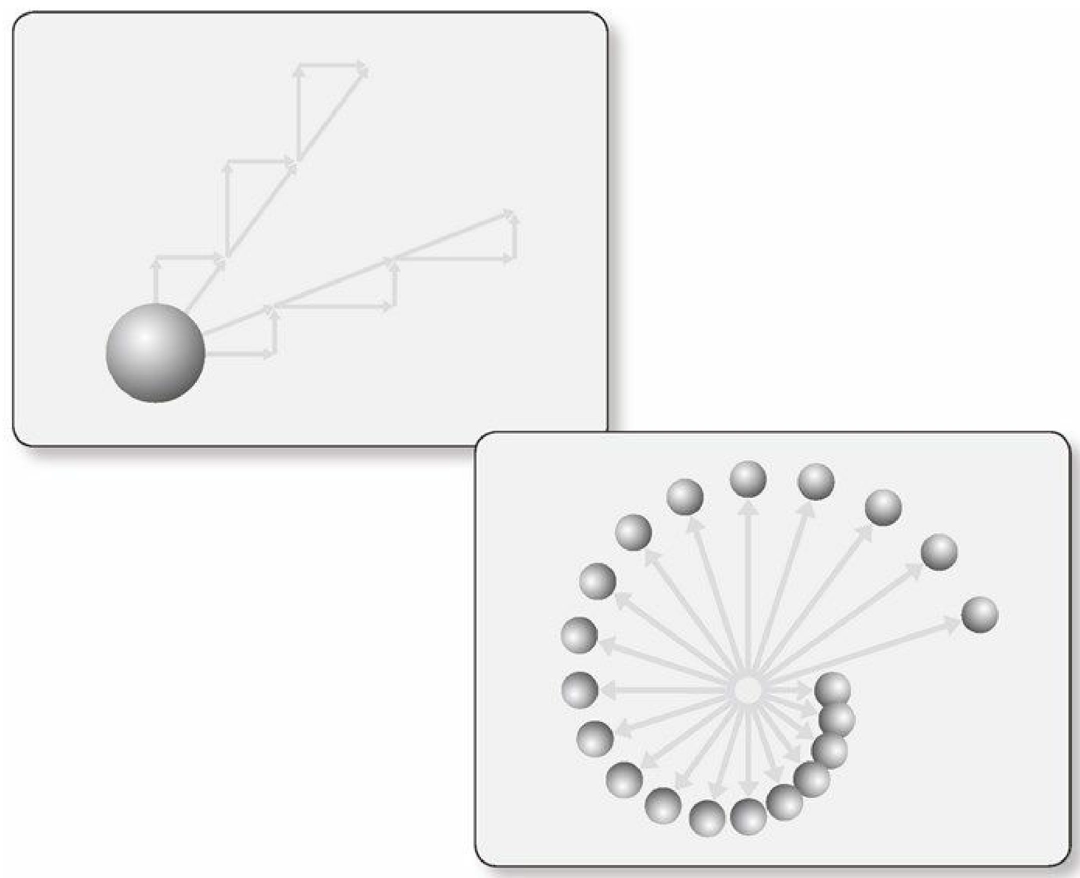
按下的情况，而没有考虑到同时按下三个方向键的情况。比如左、上、下三个方向键同时被按下时，上下的运动被抵消，物体不会向垂直方向运动，只会向左方向运动，但是由于上下方向键处于被按下的状态，所以物体在左方向的运动速度会变为平时的 $\frac{1}{\sqrt{2}}$ 倍。也就是说，我们无意中创造了一个游戏秘技，可以通过特殊操作让物体运动得比平时更慢，即同时按“上与下”或“左与右”这样相反的方向键。但是这仅限于可以不受限制地同时按任意个键盘的方向键或其他按键的环境。而考虑到多数游戏使用的都是游戏手柄或游戏摇杆等，这些设备从物理上就无法同时输入相反的方向，自然也就不会存在这一秘技，因此这里省略了三个方向键同时输入时的程序处理。如果对这部分程序心存疑虑，那就不妨自己努力打造一个不会轻易被秘技搞坏的游戏吧。

-

  - ## 1.3　让物体沿任意方向运动

Key Word　三角函数、正弦、余弦、弧度

RANK/normal



我们已经学习了让物体沿水平、斜方向运动。这次让我们再进一步，来学习如何让物体沿任意方向运动吧。

接下来，我们一起学习如何让物体以任意速度、沿任意方向运动。1.1 节介绍了如何让物体沿水平方向运动，1.2 节的后半部分介绍了如何让物体沿 45 度角的斜方向运动，让我们在此基

础上进一步拓展，让物体在任意方向以任意速度运动。还没有实践过物体沿水平方向及 45 度角方向运动的读者，建议先完成前面两小节的程序再开始本小节的学习。



图 **1-3-1**　使物体沿任意方向以任意速度运动的程序

假设物体的运动速度为 v，那么物体沿任意方向的运动一般都可以被分拆为在 x 轴、y 轴两个方向上的运动。假设分拆后物体在 x 方向上的速度为 v_x，在 y 方向上的速度为 v_y，物体运动方向与 x 轴的夹角为 θ（参考图 1-3-2）。

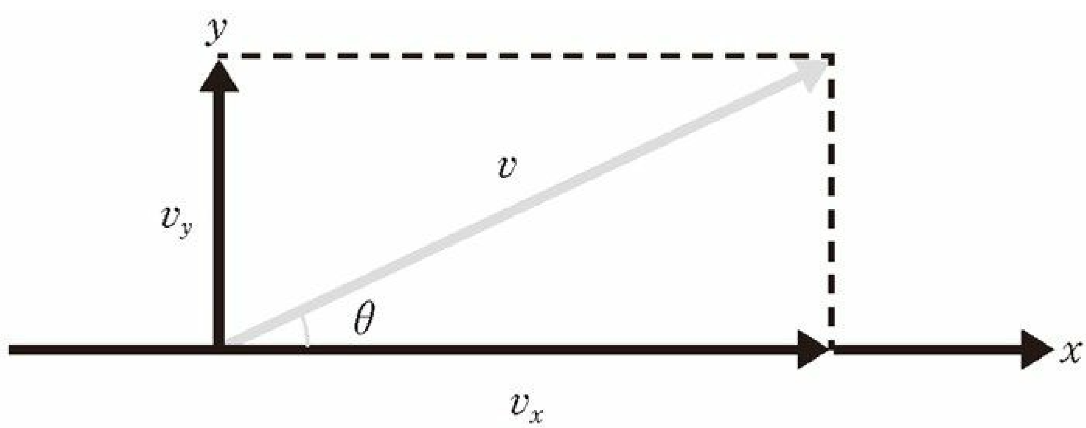


图 **1-3-2**　表示物体运动的元素

实际在画面上显示物体时必须要有 *x* 坐标和 *y* 坐标，计算坐标就需要 v_x 和 v_y，而这两者则 可以通过物体的速度 *v* 与物体运动方向与 *x* 轴的夹角 θ 计算得到。

- 让物体沿 **30** 度角方向运动的程序

  我们将上述计算程序化，制作一个让物体沿相对 *x* 轴 30 度角方向运动的示例程序，请参考 Movement_3_1.cpp（只是程序中的 *y* 坐标是基于电脑画面的，与数学中常用的坐标正好相反（向下为 +），所以请注意这里应该是向斜下方运动）。这个程序中决定物体运动的是 InitCharacter 函数中的以下 3 行代码（PI 为圆周率）。

  代码清单 **1-3-1**　沿相对于 **x** 轴 **30** 度角的斜方向运动（**Movement_3_1.cpp** 片段）

```
025  |      fAngle = PI / 6.0f;
026  |      vx = PLAYER_VEL * cosf( fAngle );          // 设置
027  |      vy = PLAYER_VEL * sinf( fAngle );
```

  程序中突然出现了正弦（sin）余弦（cos）的三角函数，为什么会用到它们呢？我们来详细 说明一下。首先可以明确的是 *x*、*y* 方向上的速度 v_x、v_y 与物体的实际速度 *v* 是成一定比例的。 也就是说，当运动方向的角度一定时，比如当 *v* 变为原来的 2 倍时，v_x 和 v_y 也应当变为 2 倍。把这种比例关系写成具体的算式，就可以得到

$$\begin{cases} v_x = a \cdot v \\ v_y = a \cdot v \end{cases}$$

这里的 a 、b 均为常数，称为比例系数 。比如系数 a 代表 v 增加 1 时，v_x 增加 a 。大家可能会觉 得不好理解，这里举一个具体的例子。比如，当物体在水平方向运动时，

$$\begin{cases} v_x = v \\ v_y = 0 \end{cases}$$

也就是说，此时的 a =1、b =0，并且物体的运动方向正好是 x 轴的方向，所以物体运动方向的 角度 θ 为 0。如果物体沿斜 45 度方向运动的话，速度正好会变为原来的 $\sqrt{2}$ 倍。

$$\begin{cases} v_x = \dfrac{1}{\sqrt{2}}v \\ v_y = \dfrac{1}{\sqrt{2}}v \end{cases}$$

上式的计算过程可以参考 1.2 节的后半部分。即如果物体运动方向的角度 θ 变成 45 度，那就是 $a = \frac{1}{\sqrt{2}}$ 、$b = \frac{1}{\sqrt{2}}$ 。也就是说，当 θ 为 0 时，a =1、b =0；当 θ 为 45 度时，$a = \frac{1}{\sqrt{2}}$ 、$b = \frac{1}{\sqrt{2}}$ 。

那么当角度 θ 取任意值时，a 与 b 的值将会如何变化呢？为了得出结论，首先可以写出常数 a 与 b 必定满足的条件。由于 v_x 和 v_y 最终会合成速度 v ，因此它们也满足勾股定理（参考图 1-3-3）。

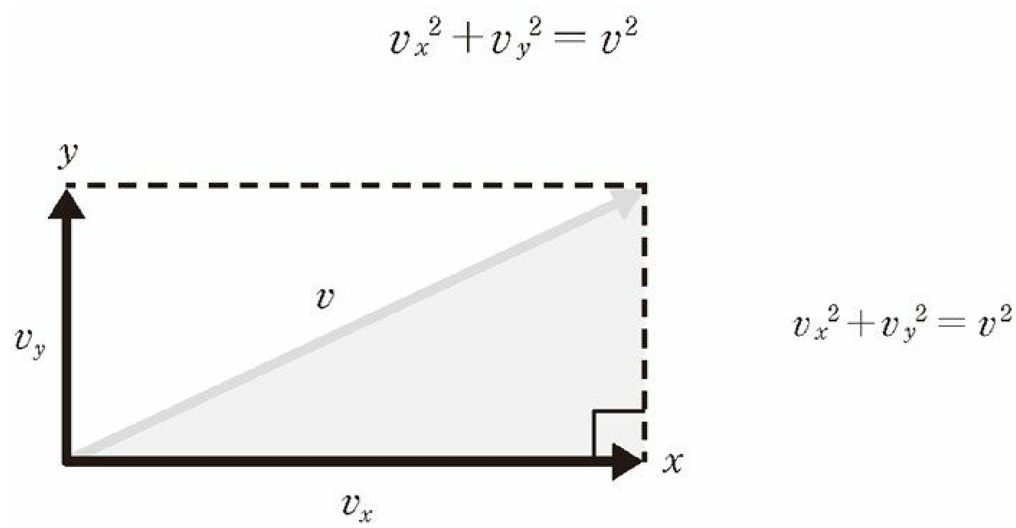


图 **1-3-3**　对速度套用勾股定理

然后将$v_x = a \cdot v$ 、$v_y = b \cdot v$ 的关系代入，得到

$$(a \cdot v)^2 + (b \cdot v)^2 = v^2$$
$$a^2v^2 + b^2v^2 = v^2$$

等式两边同时除以 v^2 ，得到

$$a^2 + b^2 = 1$$

很明显可以看出，上文中物体在水平、斜 45 度角方向运动时的 a 、b ，都满足上面的等式关系。

让我们再仔细看看这个等式，它与半径为 1 的圆（即单位圆）的公式是完全一样的。即将 a 作为横轴坐标、b 作为纵轴坐标所得到的点连接起来，正好可以画出一个半径为 1 的圆。由于 v_x 、v_y 分别是 v 乘以 a 、b 得到的，而 v_x 、v_y 所决定的物体移动方向正好与 a 、b 所做的单位圆上的点的方向相同，因此绘制出的角度 θ 将如图 1-3-4 所示。

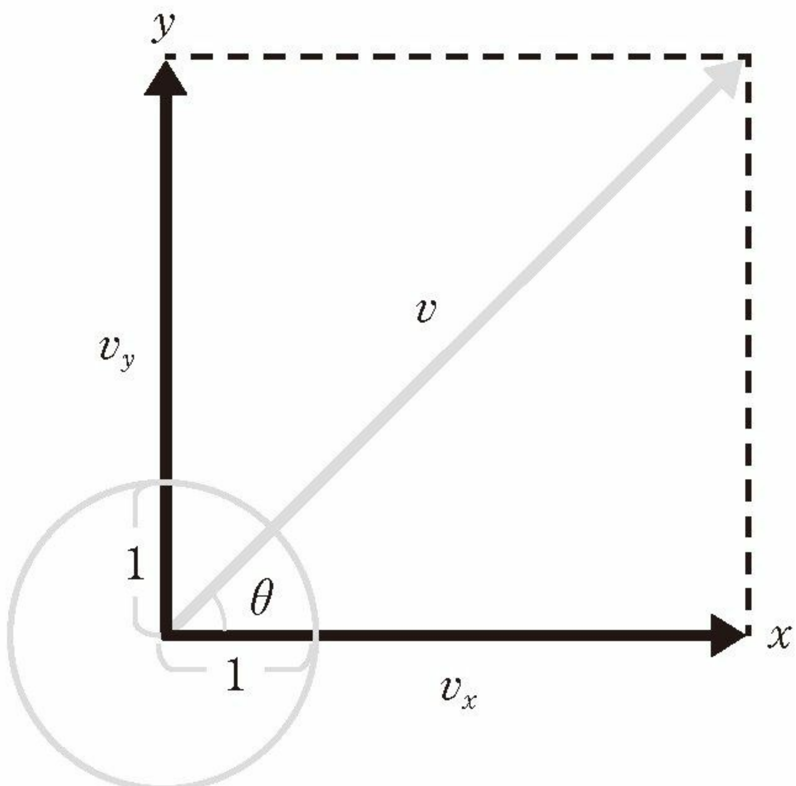


图 **1-3-4**　单位圆与角度 ***θ***

也就是说，a 与 b 分别代表单位圆上角度 θ 的点的 x 坐标与 y 坐标。根据三角函数的定义，a 与 b 可以表示成如下等式。

$$\begin{cases} a = \cos\theta \\ b = \sin\theta \end{cases}$$

看到这两个等式，是不是会想问这是为什么呢？其实这就是三角函数中余弦函数、正弦函数的定义。即角度 θ 在单位圆上的位置的 x 坐标称为余弦，y 坐标称为正弦，这是由数学家所 定义的。从上式可以推导出

$$\begin{cases} v_x = \cos\theta \cdot v \\ v_y = \sin\theta \cdot v \end{cases}$$

由此最终写出了示例程序中的语句。

只是示例程序 Movement_3_1.cpp 中仍然存在不明之处。

```
fAngle = PI / 6.0f;
```

即物体运动方向的角度被置为了 $\frac{\pi}{6}$，也就是本小节一开始所说的 30 度角方向。这是由于计算机中的三角函数（正弦余弦等）所能接受的角度，并不是以角度制 为单位的（一周有 360 度）。30 度是一周的 $\frac{1}{12}(=\frac{30}{360})$，那么程序中就必须以 $\frac{\pi}{6}$ 这样的形式为单位来表示 $\frac{1}{12}$ 周。稍加计算就能得到，$\frac{\pi}{6}$ 的 12 倍，即 2π 可以表示一周。如果有人质疑是否真的是这样，可以将程序中的 fAngle 从 $\frac{\pi}{6}$ 更改为 2π，即将运动的角度指定为 0，这样就可以看到物体会沿水平方向运动（画面右方向）。

事实上，这种将一周表示为 2π 的度量单位，称为弧度制。弧度代表半径为 1、圆心角为 θ 的圆弧，其弧长为 θ（参考图 1-3-5）。

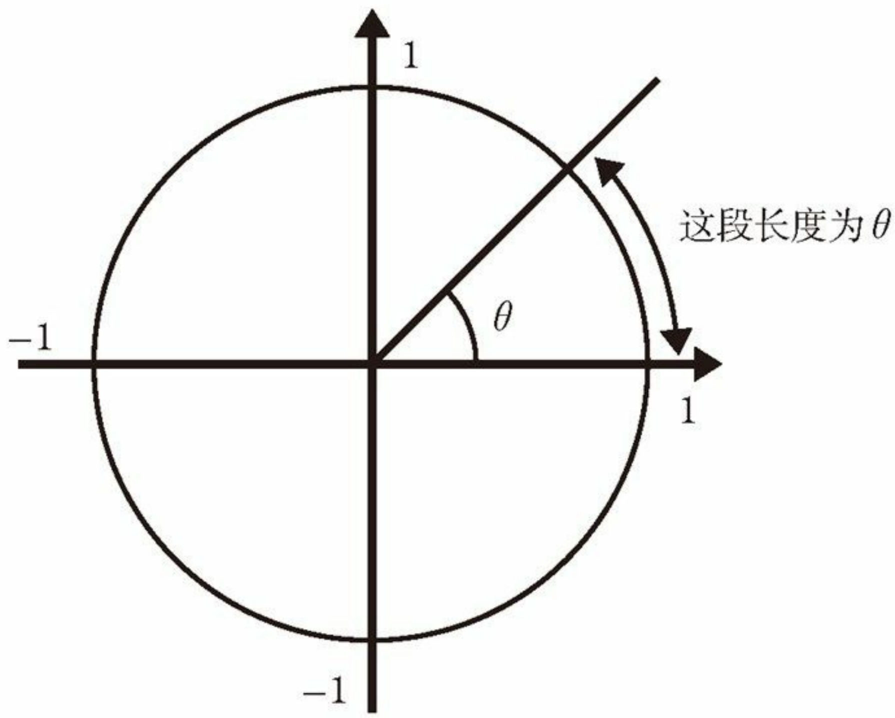


图 **1-3-5**　弧度

因此完整一周的角就可以用半径为 1 的圆（单位圆）的周长来表示。圆周长计算公式为 $2\pi r$，一周的周长为 $2\pi\times1=2\pi$，也就是一周的角的弧度值。

那么，为什么计算机中的正弦余弦函数不使用看上去比较亲切易懂的角度，而用了看来不够直观的弧度呢？其实在近代数学中，三角函数中几乎都使用弧度，很少会使用角度。之所以使用弧度而不是角度作为角的度量单位，主要是关系到微积分的一些问题。如果坚持不使用弧度的话，三角函数中与微积分相关的一些公式几乎都无法使用，还可能会引发一些严重的问题。特别是游戏中会有与物理相关的公式，稍复杂的情况都会出现微积分的计算，因此读者朋友们在程序中涉及角的度量时，请务必使用弧度而不要使用角度。如果在一个程序中同时使用弧度和角度，就会非常容易出现 BUG 及各种问题，因此将角的度量单位无条件统一为弧度这一原则，应当被无条件地贯彻下去。

POINT　请将角的单位统一为弧度。

- 使用弧度的程序

既然提到了弧度的话题，就来写个小程序实践一下吧。Movement_3_1.cpp 中物体每次出现都会向同一方向运动，这里做一个小修改让物体每次出现时的运动方向都不同。为此对 MoveCharacter 做如下改动，如代码清单 1-3-2 所示（Movement_3_1a.cpp 片段）。

代码清单 **1-3-2**　使用弧度让物体的运动方向每次都不同（**Movement_3_1a.cpp** 片段）

```
int MoveCharacter( void )                          // 每帧调用
{
    x += vx;                                       // 实际运动
    y += vy;

    // 运动到画面外时回到初始位置
    if ( ( x < -CHAR_WIDTH )  || ( x > VIEW_WIDTH ) |
        ( y < -CHAR_HEIGHT ) || ( y > VIEW_HEIGHT ) )
    {
        x = ( float )( VIEW_WIDTH - CHAR_WIDTH ) / 2.0
        y = ( float )( VIEW_HEIGHT - CHAR_HEIGHT ) / 2
        fAngle += 2.0f * PI / 10.0f;              // 增
        if ( fAngle > ( 2.0f * PI ) ) fAngle -= 2.0f *
        vx = PLAYER_VEL * cosf( fAngle );
        vy = PLAYER_VEL * sinf( fAngle );
    }

    return 0;
}
```

这个程序的关键点在于物体移出画面外时的处理。

```
        fAngle += 2.0f * PI / 10.0f;              // 增
        if ( fAngle > ( 2.0f * PI ) ) fAngle -= 2.0f *
```

第一行的作用是，当物体移出画面时使运动的角增加 $\frac{1}{10}$ 周。因此物体出现 10 次后，运动方向正好经过了一周。

这里将每次角变化的量写为了 2.0f * PI / 10.0f。数学好的人肯定会说，与其写成 2.0f * PI / 10.0f，直接写成 PI / 5.0f 不是更简单吗？如果从普通的数学题的角度来说，这确实可以简化为 PI / 5.0f。但是这里其实是有意而为之的，因为这样的写法可以一目了然地知道每次角增加的量是一周的几分之一（换句话说就是经过几次可以转一周）。书写为 2.0f * PI / 10.0f 时，不用特别计算也能知道 2π 为一周，将其 10 等分后，10 次即满一周；而如果是 PI / 5.0f，就需要去心算，一周的弧度是 2π ，2π 除以 PI / 5.0f 是多少等等，一点也不直观，理解起来也容易产生问题。程序代码不光是为了计算机，更要考虑读代码的人，因此这里特意将参数写成了 2.0f * PI / 10.0f。

如果再多考虑一些，写成 (2.0f * PI) / 10.0f 的话是不是更友好呢？众所周知，括号是可以改变运算顺序的，比如 1.0f + 2.0f * 3.0f 与 (1.0f + 2.0f) * 3.0f 的结果是不一样的。然而 (2.0f * PI) / 10.0f 与 2.0f * PI / 10.0f 的计算结果并没有什么不同（这里暂不考虑运算精度），因为 (2.0f * PI) / 10.0f 与 2.0f * PI / 10.0f 的运算顺序本来就是一样的，对于计算机来说，怎么书写都没有什么影响。但是如果写成 (2.0f * PI) / 10.0f，2.0f * PI 就被人为地划分为了一个整体，代表一周的弧度，对于人类来说则更加容易理解，因此虽然只是多了一对括号，但却让程序更加自然易懂了。

> **POINT**　编程时不仅要考虑到计算机，还要考虑到读代码的人。

让我们再回到代码。请大家看一下程序的关键部分的第 2 行。

```
045  |        if ( fAngle > ( 2.0f * PI ) ) fAngle -= 2.0f *
```

这一句有什么作用呢？如注释所示，这行语句会在弧度增加到比一周大时，减去一周的弧度。其实弧度增加或减少一周，等于旋转了一周后又回到了原位，其代表的夹角方向并没有改变，从数学角度来说，这一行代码是没有任何意义的。事实上即使将这行删除，至少在这个程序中，也会得到相同的运行结果。

那么这行代码真的只是摆设吗？并不是这样的。这行的意义在于控制角的弧度始终在 $0 \le \theta \le 2r$ 的范围内，不让角 θ 无限制地增大下去。确实从数学角度来说，角 θ 无论多大（即弧 度增加若干周）都没有问题。比如 2π 、4π 、6π ……全部表示同样的角，2.5π 、4.5π 、6.5π …… 全部等同于 $\frac{\pi}{2}$ （90 度）的角。数学中一个角无论多大都不会有问题，而实际上计算机用一个极大值计算正弦余弦时却可能出问题。

因为角的弧度值在计算机内部是以浮点数 的形式表示的，即形如 1.2×10^2 这种指数形式的数字。浮点数只用很少的位数（= 很少的内存容量），就可以表示 100 000 这样的大数。与此相对，浮点数存在计算绝对值较大的数时会丢失精度的问题。比如计算 10 与计算 100 000，计算误差会相差一万倍。因此计算机在计算含有小数点的数字时，会受到绝对值较大的数的影响丢失精度。如果轻视上面的问题，程序在循环中一次次增加弧度时，随着循环次数的增多，弧度增大到某个值之后精度就会开始丢失，最终程序运行的结果可能会与预期完全不一样。因此除非有特殊需要，我们在编程时都要注意避免产生一个极大的弧度值。此外，计算机的三角函数在处理一个很大的弧度值时，其运算时间也可能会加长，这里就不再具体论证了。总之我们要尽可能地控制 θ 的取值范围在 $0 \le \theta \le 2\pi$ （或者 $-\pi \le \theta \le \pi$ ）之内。

POINT 计算机计算不可避免的会有误差，绝对值越大误差也越大。因此应该尽可能地使用绝对值小的数字进行计算。

-
  -

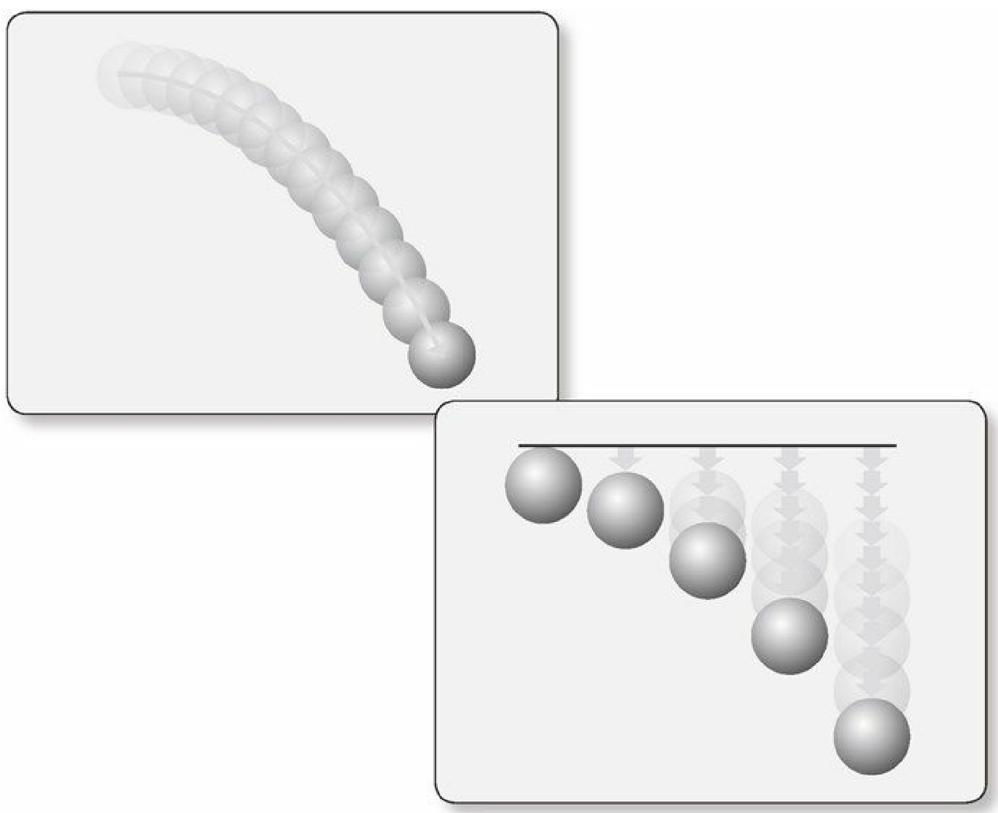
▪ 1.4　在物体运动中加入重力

Key Word　抛物运动、重力加速度、计算误差、积分



前半部分


后半部分



在现实世界中，物体会受到重力向下的作用力。本节就让我们来学习如何将重力应用到游戏世界中，让运动表现得更加真实吧。

本小节中，我们将讲解如何通过给物体施加重力来更加真实地表现物体的弹跳等运动。物体在重力作用下的运动称为抛物运动，Movement_4_1.cpp 为将抛物运动简单地用

编程语言编写出来的示例程序。

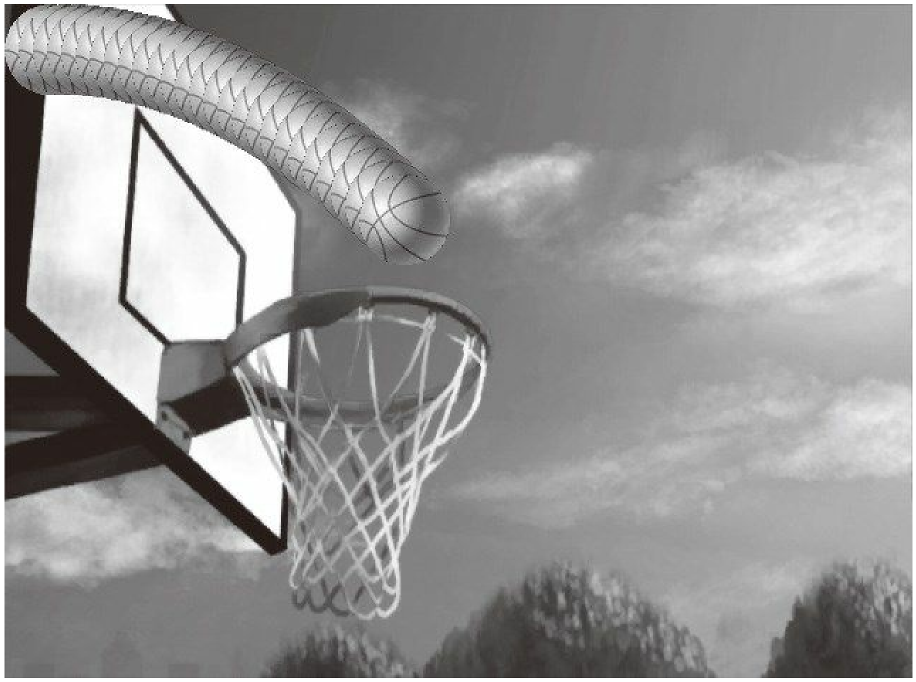


图 **1-4-1**　抛物运动的程序

其中最关键的是 MoveCharacter 中的以下 2 行。

```
    y += vy;            // 对位置加入速度
    vy += GR;           // 为速度加上加速度
```

其中 *vy* 是物体在 *y* 方向的速度，GR 为重力加速度 。所谓重力加速度，即做抛物运动的物体被施加的加速度。为了进行下面的加速度相关的话题，首先来确认一下速度与加速度的关系。速度可以表达为

$$\text{速度} = \frac{\text{移动距离}}{\text{时间}}$$

所以移动距离其实就是位置的变化量，可将其写成 Δx ，

希腊字母 Δ 代表其标记的值为变化量。同理，时间也等同于时刻的变化量，将其书写为 Δt 。那么速度 v 就可以表达为

$$v=\frac{\Delta x}{\Delta t}\cdots\cdots①$$

这是位置与速度的关系式，其实这个关系式也适用于速度和加速度。假设加速度为 a ，速度的变化量为 Δv ，则有

$$a=\frac{\Delta v}{\Delta t}\cdots\cdots②$$

由上面的式子可以知道，每单位时间位置的变化量是速度（根据式①），每单位时间速度的变化量是加速度（根据式②）。那么将上面的等式再做一下变形，就可以得到

$$\Delta x=v\Delta t$$
$$\Delta v=a\Delta t$$

时间 Δt 的单位为帧，在一般的游戏中，1 次循环经过的时间为 1 帧，即 $\Delta t=1$。代入上式得到

$$\left.\begin{aligned}\Delta x&=v\\\Delta v&=a\end{aligned}\right.\cdots\cdots③$$

而由于 Δx 是 x 的变化量，假设当前的 x 为 x_n ，上一个 x 为 x_{n-1} ，则有

$$\Delta x=x_n-x_{n-1}$$

同理，Δv 是 v 的变化量，假设当前的 v 为 v_n ，上一个 v 为 v_{n-1} ，同样有

$$\Delta v=v_n-v_{n-1}$$

那么式③就可以表示为

$$x_n-x_{n-1}=v$$
$$v_n-v_{n-1}=a$$

然后将 x_{n-1} 与 v_{n-1} 移项到右边，得到

$$
x_n = x_{n-1} + v \\
v_n = v_{n-1} + a
$$

将上式中的 *x* 置为 *y*，*a* 置为 GR，就得到了程序中的语句。只是单凭上面的讲解理解起来可能有点难，请仔细观察程序的运行以帮助理解。

自然界中的物体都会受到重力作用而产生向下的加速度，这个加速度的值是固定的，程序中的常数 GR 就表示该加速度。GR 的值在每帧都会被加到 *vy* 变量中，所以 *y* 方向的速度 *vy* 会逐帧增加，像 0、GR、2GR、3GR……这样，每帧都会增加 GR，同时每次增加的速度还会被加到位置 *y* 中去，这样最终程序运行的结果就是带有加速度的抛物运动了。

需要注意的是这里我们用到的重力加速度 GR，并不是自然界中平时使用的值 9.8。众所周知的重力加速度值 9.8 是真实的地球上的重力加速度，以 m/s^2（米 / 平方秒）为单位，并不能在 计算机的虚拟空间中适用。计算机所使用的重力加速度单位是特殊的 $\mathbf{dot/F^2}$（像素 / 平方帧），请注意区别。

- 抛物运动在斜抛中的应用

  接下来，让我们将抛物运动从平抛扩展到斜抛。为此将 InitCharacter 函数中的

```
    y = 0.0f;           // y方向的初始位置
    vy = 0.0f;          // y方向的初始速度
```

  更改为（Movement_4_1a.cpp）

```
    y = 200.0f;          // y方向的初始位置
    vy = -10.0f;         // y方向的初始速度
```

物体被向上抛出后，在重力作用下会改变运动方向，变为向下掉落。这与游戏中角色进行跳跃的动作是一致的，可以类推到很多场景中。

那么到此是不是就可以结束了呢？很遗憾还不能。我们到目前为止得出的结论，对于那些物体运动不是至关重要的程序来说是足够用的，比如业余时间制作的游戏中角色的动作，或者大量粒子飞溅的效果（particle）等。但如果是用于商业游戏，特别是动作比重非常大，如动作游戏中角色的运动，上面的结论是有缺陷的。

写了这么半天，结果却说得出的结论有缺陷，可能会有读者感到不满吧。其实所谓的缺陷，并不是说我们之前的结论是错误的，只是告诉读者上面的结论中存在不够准确的部分。如果用更接近数学上的语言来说，就是在之前的一些算式中，只是粗糙地使用了近似值，经过很长时间之后计算结果可能会变得不准确。具体来说，比如

$$\text{速度} = \frac{\text{移动距离}}{\text{时间}}$$

这一部分。这个算式小学生应该都学过的吧，那么有些思维严谨的孩子会不会多想一些问题呢？比如：分母为时间，但如果加速的话，这段时间内的速度也会改变，这样一来上面的等式不是就有问题了吗？是的，有这样的疑问一点也不奇怪。因为上面的等式中要计算的速度，本来就会根据计算时所取的时间的不同而变化。而这个“速度 = 距离 / 时间”的等式，也只有在速度恒定时才成立，有加速度时就不正确了。有

加速度时，就必须考虑到在 Δ*t* 这段有限的时间内，速度也是不断变化的（参考图 1-4-2）。

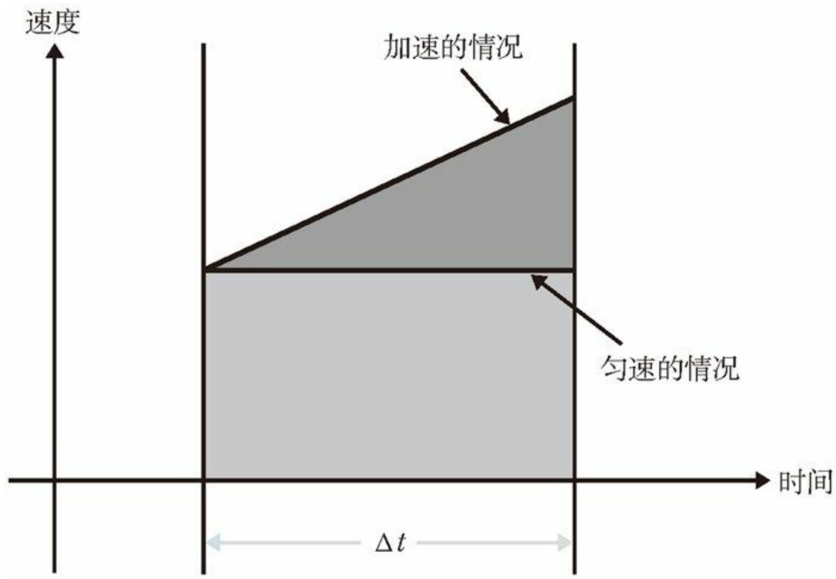


图 **1-4-2**　匀速与加速时经过相同时间后产生的差异

在有加速度时，无论取 Δ*t* 时间前的速度，还是取 Δ*t* 时间后的速度，用上面的算式计算出的某时刻的位置都是不正确的，因为速度是在不断变化的。

那么在有加速度时怎样正确地计算位置呢？我们需要更改一下思考方式。设想有无限小的时间 ，然后将所有在无限小的时间内行进的距离（距离也是无限小的）都加起来就可以了。也就是说，我们最终会将无数个无限小的数加起来（听起来有些像哲学上的问答）。当然计算机是无法处理无限小的数的，也不可能真的去将无数个数相加。所以在实际的程序中，都是人类使用某些算法计算无限小的数字后，直接将计算结果添加到程序中的。

可能很多读者已经明白了，这个所谓的将无数个无限小的数字相加的操作，在数学上称为积分 。是的，就是有名的微积分中的积分。听到微积分，可能不少人都会多少有些抵触情绪吧。其实包括我自己都不太想去碰及，但是为了达到目的，也只好硬着头皮上

了。因为积分正好可以解决被施加了重力的状态下的问题。不过好在这里并不会涉及微积分比较深入的部分，如果想比较深入地学习微积分，少则也要花两三年时间吧。我们的目的不是要成为数学家，而是要制作游戏，所以只要囫囵吞枣地学会使用伟大先哲们总结出的公式就可以了。

假设位置为 x 、速度为 v 、加速度为 a 、经过时间为 t ，将会有以下关系式成立。

$$x = \int v \mathrm{d}t$$

$$v = \int a \mathrm{d}t$$

这里出现的 $\int f(x) \mathrm{d}x$ ，是函数 $f(x)$ 关于变量 x 的不定积分。对这部分知识不够了解的读者，可以参考第 6 章。

让我们回到刚才的话题，在已经论证过的等式

$\Delta x = v \Delta t$

中加入无限小的概念。由于上例中 y 方向上被施加了 GR 的重力加速度，因此将 GR 表示为 G，则有

$$y = \int v_y \mathrm{d}t$$

$$v_y = \int G \mathrm{d}t$$

由后一个式子可以得到

$$\begin{aligned} y &= \int G \mathrm{d}t \\ &= Gt + C_1 \end{aligned}$$

C_1 称作积分常数 ，代表 t =0 这一时刻的速度，也就

相当于物体的初始速度 。而由前一个式子 =可以得到

$$
\begin{aligned}
y &= \int v_y \mathrm{d}t \\
&= \int (Gt + C_1) \mathrm{d}t \\
&= \frac{1}{2}Gt^2 + C_1 t + C_2
\end{aligned}
$$

C_2 这个积分常数，是 t =0 时的 y 坐标，即物体的初始位置 。我们最终得到的这个式子，可能看过大学数学课本的人都会有点印象，其实这就是使用积分所推导出的结果。

那么与使用积分的正确方法相比，我们一开始采用的通过循环在位置中叠加速度，并在速度中叠加加速度的方法，究竟有多大的偏差呢？简单起见，我们将初始速度和初始位置都置为 0，那么在使用积分的方法中的等式

$$y = \frac{1}{2}Gt^2 + C_1 t + C_2$$

中，就相当于 $C_1 = 0$、$C_2 = 0$，得到

$$y = \frac{1}{2}Gt^2$$

而在循环中每次叠加一个数字，这在数学上叫作级数 。比如 t 时刻的速度 v_y 可以写成

$$
\begin{aligned}
v_y &= \sum_{i=1}^{t} G \\
&= Gt
\end{aligned}
$$

到目前为止，两种方法计算速度的结果还是一样的。但是很遗憾，之后计算位置 y 时结果就不正确了。

$$
\begin{aligned}
y &= \sum_{i=1}^{y} v_y \\
&= \sum_{i=1}^{t} G(i-1)
\end{aligned}
$$

在上式中，之所以将 Gt 表示为了 $G(i-1)$，是因为 y 在加了 v_y 之后，由于需要给 v_y 加上加速度，计算 y 时所使用的 v_y 的更新被延后了 1 次。于是有

$$
\begin{aligned}
y &= \sum_{i=1}^{t} G(i-1) \\
&= G \cdot \frac{1}{2} t(t-1) \\
&= \frac{1}{2} G t^2 - \frac{1}{2} G t
\end{aligned}
$$

相比使用积分时正确的计算结果，

$$
y = \frac{1}{2} G t^2
$$

结果中少了 $-\frac{1}{2} G t$ 。也就是说，对位置叠加速度，再对速度叠加加速度这一简单算法，在程序中随着时间的推进，误差会变得越来越大。如果真正的游戏中出现了这种误差，游戏的玩家很可能会认为游戏有问题，甚至会听到这样的抱怨：“这个角色的动作有问题啊，移动起来总是差了 3 个像素，真是个烂游戏！”

上面的例子可能有点吹毛求疵，但是如果忽视这小小的误差真的会造成很多问题。比如网球、高尔夫球、棒球等球类比赛中球的运动总是至关重要的，球的旋转以及空气摩擦的影响等因素都必须考虑在内，如果有误差就可能会影响球的运动轨迹。在动作游戏中，误差会引起帧速率不固定，可能会使运算结果无法重现，例如因运算时机受到误差影响而丢掉一帧的运

算。此外有多台机器同时开发一个游戏时，如果不注意误差，就会出现配置好的机器和配置差的机器的帧速率不一致的情况。而计算机的运算速度也不是恒定的，必须保证运算时间有变化时物体运动的速度也不受影响。如果帧速率无法保证，虽然每一帧的计算误差极小，但最终表现出的差异也会很惊人，有时候甚至会让物体彻底偏离运动轨道，特别是在动作游戏中，一点微小的动作偏差都会影响游戏性，最终演变成大问题。因此为了保证计算的精度，希望大家尽可能地在程序中使用积分进行运算。

- 使用积分制作的抛物运动程序

使用积分实现的抛物运动程序，请参考示例程序中的 Movement_4_1b.cpp。在该程序的 MoveCharacter 函数中，

```
    x = vx * t;                                  /
    y = 0.5f * GR * t * t + vy * t + 200.0f;     /
```

这一部分是应用积分计算的结果求 *x* 方向、*y* 方向的位置。如果用通常的算式写出来就是

$$x = v_x t$$
$$y = \frac{1}{2}Gt^2 + v_y t + 200$$

这个程序的特点是，只需要知道物体运动过程中的具体时刻，就可以直接算出物体的位置。这样处理能够使误差不会随着时间慢慢增大，所以即使帧速率不固定，物体也可以沿同一轨道行进。不过使用积分算法的程序 Movement_4_1b.cpp，与使用叠加算法的程序 Movement_4_1a.cpp 相比，用肉眼应该无法看出什么区别，可以实际运行两个程序对比一下。

那么是不是使用上面的积分来精确计算轨道就万无一失了呢？很遗憾这是不可能的。如果地球上只有重力一种力的话还没什么问题，但是物体运动往往会受到多个复杂的力的共同作用，想精确求得物体每一个瞬间的轨道几乎是不可能的，这种情况下只能根据数据计算物体近似的运动轨道。

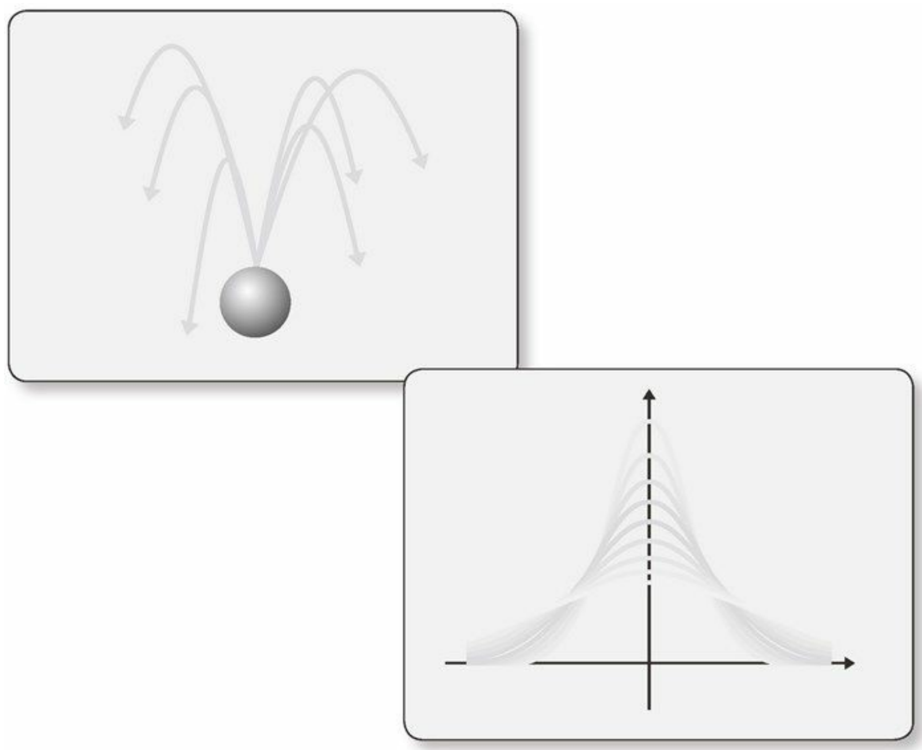
1.5 物体随机飞溅运动

Key Word 随机数、均匀随机数、正态分布



前半部分 

后半部分



本小节将实现大量物体随机飞溅的效果。随机飞溅虽然是常见的现象，但实现起来却没那么容易哦。

本小节中，让我们来学习大量物体以随机初速度飞溅的运动。比如火山喷发、烟花、摩擦迸出的火花等状况都可以套用这种运动。由于物体飞溅运动其实也是

在进行抛物运动，所以建议大家掌握 1.4 节中抛物运动的程序后，再来阅读本小节。

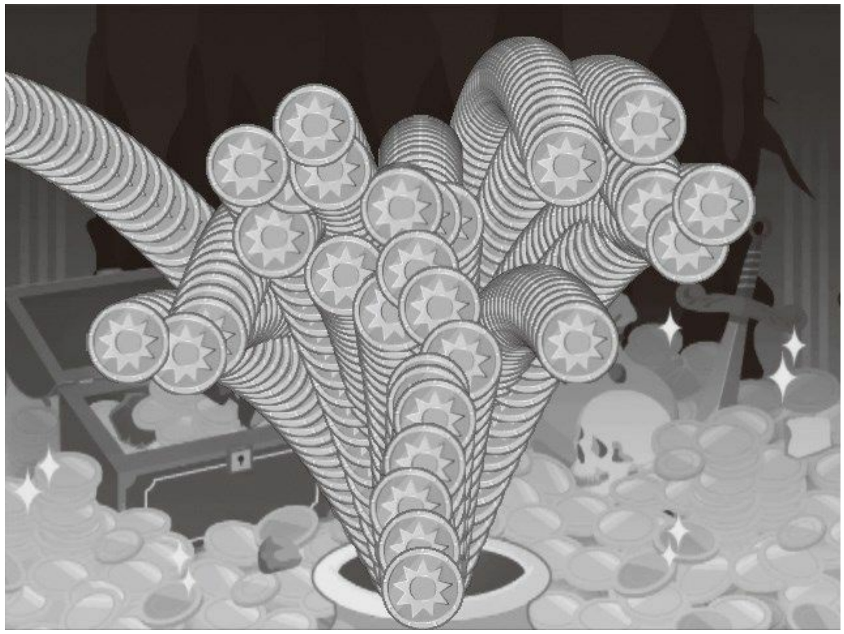


图 **1-5-1**　物体随机飞溅运动的程序

将物体随机飞溅运动程序化之后的示例程序为 Movement_5_1.cpp。这个程序的重点是如何产生随机的初速度，具体是以下 4 行（代码清单 1-5-1）。

代码清单 **1-5-1**　产生随机速度的部分（**Movement_5_1.cpp** 片段）

```
    Balls[i].vx = rand() * VEL_WIDTH / ( float )
        - VEL_WIDTH / 2.0f;                    /
    Balls[i].vy = rand() * VEL_HEIGHT / ( float )
        - VEL_HEIGHT / 2.0f - BASE_VEL;        /
```

为了理解这两行代码，首先就需要理解如何产生一个从 0 到 n 范围内的随机数。在这个程序中，为了获得 0～n 的随机数，使用了以下算法。

```
rand() * n / ( float )RAND_MAX
```

为什么这样就可以得到 0～*n* 的随机数呢？这需要先明白 RAND_MAX 常数代表什么。读过一遍示例程序就可以发现，RAND_MAX 常数在这个程序中并没有定义，而是事先在 C 语言等的头文件中定义好的，代表 rand 函数所能返回的最大值。所以 rand() / (float)RAND_MAX 这个语句就可以得到 0 到 1 之间的随机数。而 rand() * n / (float)RAND_MAX 是将 0 到 1 之间的随机数再乘以 *n*，因此结果就可以得到从 0 到 *n* 的随机数。

既然 rand() / (float)RAND_MAX 得到的是从 0 到 1 的值，那么将获得从 0 到 *n* 的随机数的程序写成

```
(rand() / ( float )RAND_MAX ) * n
```

不是更容易理解吗？这里之所以写成 rand() * n / (float)RAND_MAX，是因为考虑到如果 *n* 是整型，就可以首先计算 (rand() * n) 的整型乘法，这样在速度上会更有保证。如果不小心将上面的语句更改为

```
(rand() / RAND_MAX ) * n
```

(rand() / RAND_MAX) 部分进行计算时小数点以后的

部分就会被全部舍弃，导致大部分结果都是 0，程序也就失去原有的意义了。

可能有编程经验的读者会想到使用求余的方式。确实，获得 0～*n* 的随机数时，用求余运算是比较普遍的方式，如下所示。

```
rand() % ( n + 1 )
```

其中 % 代表求余运算符，上面的语句的效果是，无论 rand 函数生成的随机数的范围是多少，只要将其产生的随机数除以 (*n* +1)，就可以得到 0～*n* 的随机数。比如用上述方法生成 0～3 的随机数时，rand 的返回值如表 1-5-1 所示。

表 **1-5-1　rand** 函数的返回值以及最终结果

A	0	1	2	3	4	5	6	7	8
B	0	1	2	3	0	1	2	3	0

A: rand 函数的返回值　B: rand() % (3 + 1) 的值

使用这个方法确实可以将 rand 函数的返回值归纳为 0～*n* 的随机数。那么为什么我们没有采用这种方式去编写之前的示例程序呢？这是有若干原因的。其中最重要的原因是，采用求余的方式获得的随机数必然只能是整数。因为所谓“求余”，就是在整数范围内取得一个余数。然而在示例程序中，如果将速度的 *x* 分量和 *y* 分量都设置为整数，向同一方向飞溅的物体就会骤然增多，导致效果显得很不自然。对此有兴趣的读者可以将上面的 4 行代码做如下变更（Movement_5_1a.cpp）。

```
    Balls[i].vx = ( rand() % ( VEL_WIDTH + 1 ) )
        - VEL_WIDTH / 2.0f;
    Balls[i].vy = ( rand() % ( VEL_HEIGHT + 1 ) )
        - VEL_HEIGHT / 2.0f - BASE_VEL;
```

虽然这种写法很简单，但通过运算结果就可以发现，向正上方飞溅的物体增多了，具有同样速度的物体也增多了，看起来多多少少显得不够自然。

再来看示例程序 Movement_5_1.cpp 中计算 vx 的地方。

```
    rand() * VEL_WIDTH / ( float )RAND_MAX
        - VEL_WIDTH / 2.0f;            // 随机
```

程序会首先产生从 0 到 VEL_WIDTH 的随机数，然后减去 VEL_WIDTH / 2.0f，最终就产生了 -VEL_WIDTH / 2.0f 到 VEL_WIDTH / 2.0f 范围内的随机数。由于一般物体随机向上飞溅时，我们都希望向左方向与向右方向产生的飞溅效果是对称的，因此这里使正负值以相同的概率产生。同理，对 vy 的计算也采用同样的方式，

```
    rand() * VEL_HEIGHT / ( float )RAND_MAX
        - VEL_HEIGHT / 2.0f - BASE_VEL;        /
```

之所以这样设置，是因为在火山喷发等场景中，通过设置一个平均速度为基础速度，然后使所有的物体飞溅都在基础速度上再随机加上一个加速度 *a*，从而就可以让整体效果显得更加真实。此时产生的随机数范围是 (-BASE_VEL - VEL_HEIGHT / 2.0f) 到 (-BASE_VEL + VEL_HEIGHT / 2.0f)。

按照上面的思路，不难想象我们可能会经常用到一个功能，即产生从 *a* 到 *b* 的随机数。为此，我们可以把这个功能提炼成函数，通过此函数来产生指定范围内的随机数。此时随机数的计算公式如下所示。

```
rand() * ( b - a ) / ( float )RAND_MAX + a
```

请读者自行确认之前的计算 vx 与 vy 的式子是否满足该公式。

- 为什么会感到不自然

  上面我们制作了一个限制随机数范围，让物体随机向上飞溅的程序，但是有些人可能会感觉其运行结果有些不自然。其实这个程序中的随机数的产生方式，与自然界中真正的物体飞溅现象相比，是有很大不同的。当然，自然界中物体飞溅的方向是 3D 的，这是其中一个原因，然而更重要的原因是，自然界中这种情况下的初速度分布并不是均匀随机数 ，而是满足正态分布 的。下面就依次说明什么是均匀随机数 和正态分布 。

  首先说明均匀随机数。所谓均匀随机数，是指所有数字的出现概率都相等的随机数，也就是之前程序使用 rand 函数所产生的随机数 [1] 。使用均匀随机数所造成的现象就是物体向正上方的概率与其他方向的概率都是相同的。

而所谓正态分布，是指随机数中有一个值取得的概率最大，其他值取得的概率比较小，具体来说，正态分布可以表示为

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{ \frac{(x-\mu)^2}{2\sigma^2} \right\}$$

其中 μ 代表均数，即最高概率出现的点，σ 则称为标准差，表示概率分散的程度（参考图 1-5-2）。

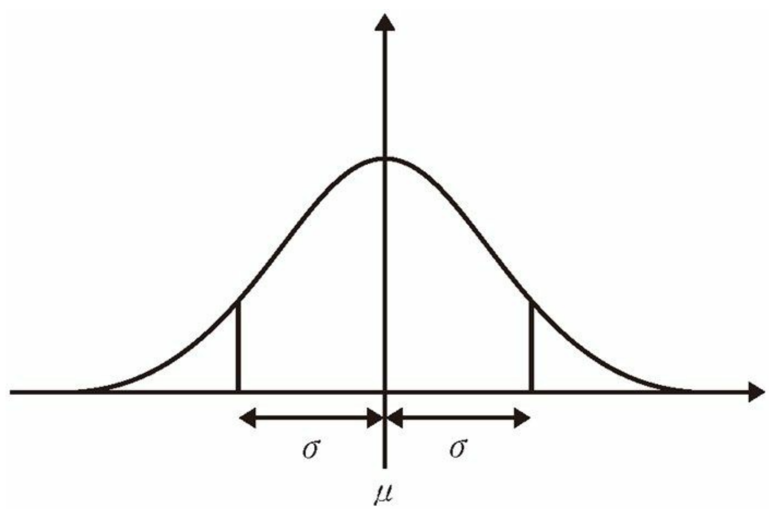


图 **1-5-2**　正态分布的图像

以自然界中的火山喷发为例，火山岩会以最高的概率向某个特定方向喷溅（一般是正上方），向其他方向喷溅的概率较低。而在 Movement_5_1.cpp 中，由于我们没有限制速度的取值范围，速度在所有方向上的取值都是等概率的，因此看上去就会觉得不太自然。

那么如果我们从自然界中学习，让速度满足正态分布，物体的运动会不会变得自然一些呢？答案当然是肯定的。但是想要在程序中实现均数为 μ、标准差为 σ 的正态分布却不是一件容易的事。C 语言中并没有提供一个方便的函数可以直接生成满足正态分布的随机数，我们只能基于产生均

匀随机数的 rand 函数，通过自己的计算来实现。但是如果要自己琢磨计算方法恐怕很有难度，因此我们需要仰仗一下先贤的智慧。

Box-Muller 算法 是一种能根据均匀分布的随机数来产生正态分布的随机数的算法。根据 Box-Muller 算法，假设 *a* 、*b* 是两个服从均匀分布并且取值范围为从 0 到 1 的随机数，我们就 可以通过下面的公式获得两个满足正态分布（均数为 0，标准差为 1）的随机数 Z_1 和 Z_2 。

$$Z_1 = \sqrt{-2\ln(a)}\cos(2\pi b)$$
$$Z_2 = \sqrt{-2\ln(a)}\sin(2\pi b)$$

等式中的 ln(*x*) 代表自然对数 函数，即以 e (=2.71828...) 为底的对数函数。上式对应的图像是在半径为 $\sqrt{-2\ln(a)}$ 的圆上取随机角度的点，点的 *x* 坐标为 Z_1 ，*y* 坐标为 Z_2 。为什么这样的 Z_1 和 Z_2 会满足正态分布呢？详细说明起来会比较复杂，也超出了本书的范围，因此在此略过。总之大家只要将 Z_1 和 Z_1 作为两个没有任何相关性的随机数去使用就可以了。

将上式实现为程序，即示例程序 Movement_5_2.cpp。这个程序首先会生成均数为 0、标准差为 1 的满足正态分布的随机数，并通过下面的方式限制随机数的范围。

代码清单 **1-5-2**　限制利用正态分布生成的随机数的范围

```
078  | Balls[i].vx = ( fRand_r * cosf( fRand_t ) )
079  | Balls[i].vy = ( fRand_r * sinf( fRand_t ) )
```

与之前程序中使用的均匀随机数不同的是，在这段程序中，例如 vx 的取值范围没有被限制在 -VEL_WIDTH / 2.0f 到 VEL_WIDTH / 2.0f 之间，

即速度超过此范围的物体也会偶尔出现。这样，我们最终就实现了自然界中的随机现象——正态分布。

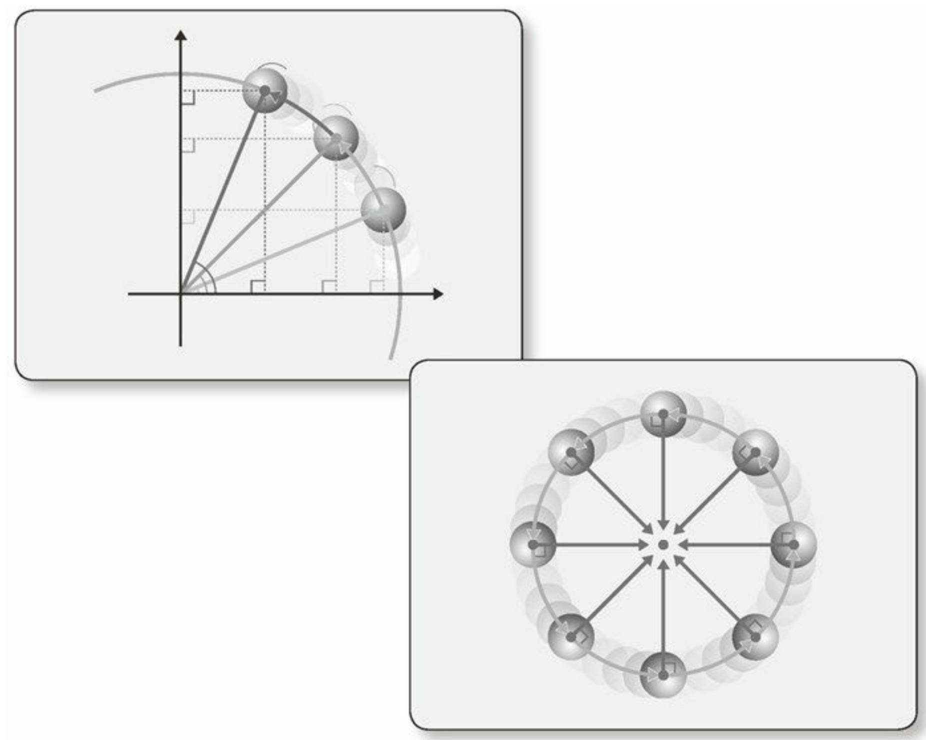
[1] 严格讲，rand 函数所产生的随机数只能称为“伪随机数”，并不一定是真正意义上的均匀随机数，只是经过将 0～RAND_MAX 范围内的值转换为 a～b 范围内的值这一处理后，虽然结果可能会和均匀随机数有所偏差，但由于对程序的影响较小，因此直接忽略了。

-
    -
        -

1.6 让物体进行圆周运动

Key Word 角速度、向心力 RANK/normal

前半部分 RANK/hard 后半部分



在本小节，我们将实现使物体以一点为中心进行旋转的圆周运动。

下面就让我们来学习如何让物体围绕一个中心点做圆周运动吧。



图 **1-6-1**　物体围绕中心点旋转的程序

将运动简单程序化后的示例程序为 Movement_6_1.cpp。这个程序中，重点部分是 MoveCharacter 中的以下 3 行。

代码清单 **1-6-1**　使物体围绕中心点旋转的程序的主要部分

```
    x = ROT_R * cosf( fAngle ) + ( VIEW_WIDTH -
    y = ROT_R * sinf( fAngle ) + ( VIEW_HEIGHT -
    fAngle += 2.0f * PI / 120.0f;           //
```

这里并没有考虑物体运动中的速度和加速度的情况（如重力作用），而是直接计算了物体的位置。具体来说，是基于以下原理计算的。首先根据三角函数的正弦余弦定义有

$$x = \cos\theta$$
$$y = \sin\theta$$

在一个以原点为中心的单位圆 (半径为 1 的圆) 上，根据角度 θ 就可以表示一个点的位置（参考图 1-6-2）。

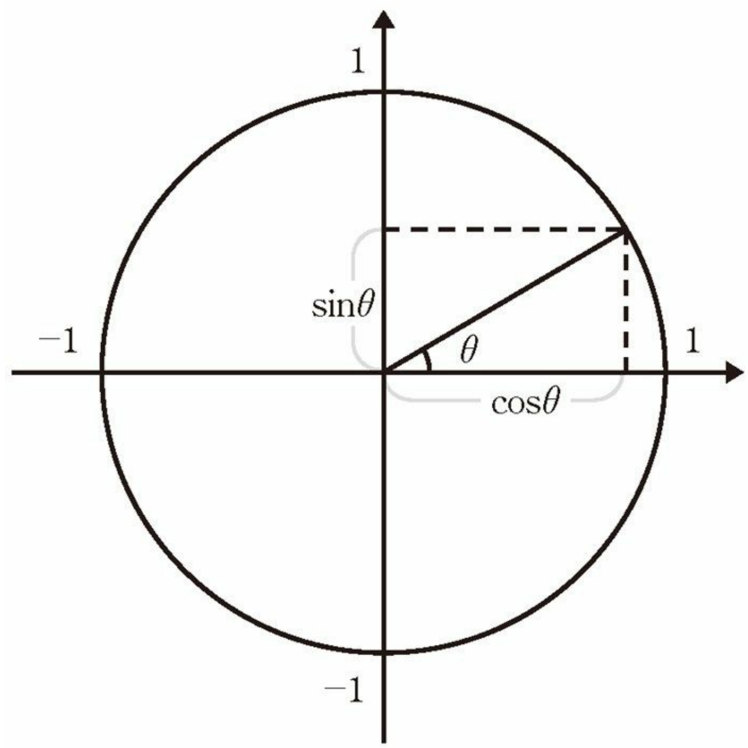


图 **1-6-2**　三角函数的定义

如果上式中 θ 的值随时间递增，就会形成以原点为中心半径为 1 的圆周运动（参考图 1-6-3）。

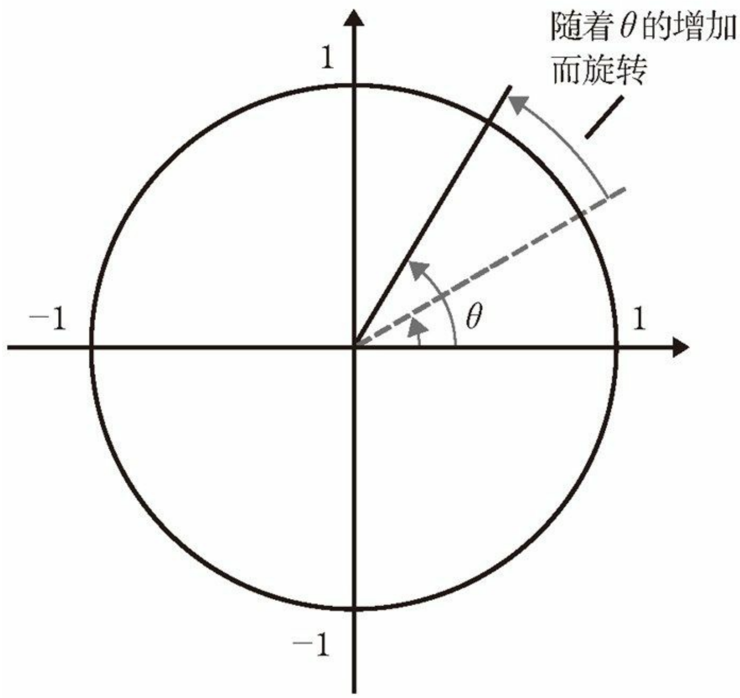


图 **1-6-3**　伴随时间进行的圆周运动

将这个圆的半径乘以 r，并将原点设置为 (x_0, y_0)，那么围绕着 (x_0, y_0) 的半径为 r 的圆周运动就为

$$x = r \cdot \cos\theta + x_0$$
$$y = r \cdot \sin\theta + y_0$$

于是就有了程序中的以下两行。

x = ROT_R * cosf(fAngle) + (VIEW_WIDTH - CHAR_WIDTH) / 2.0f;

y = ROT_R * sinf(fAngle) + (VIEW_HEIGHT - CHAR_HEIGHT) / 2.0f;

此外，

```
034  |     fAngle += 2.0f * PI / 120.0f;          // 
```

这一行决定了物体的旋转速度。在物理学中表示物体的旋转速度时常使用角速度 。角速度一般写作 ω，可以表示为 $\theta = \omega t$ 。那么角速度就满足等式 $\omega = \frac{\theta}{t}$，即单位时间内角度的变化量。

使用角速度 ω 计算虽然方便，但是究竟多长时间才能旋转一周呢？我们无法通过角速度直观地得出结论。为此就有了周期 T 的概念，$\omega = \frac{2\pi}{T}$ 即 $\theta = 2\pi\frac{t}{T}$ 。显然，经过时间 T 后，物体正好旋转一周（2π =单位圆的圆周长，正好是 T 时间所经过的角度）。因此示例程序中每次为角度 fAngle 增加 $\frac{2\pi}{120}$，就用 120 帧的时间（2 秒）完成了一周圆周运动。

用上面的方法，可以在不考虑加速度的情况下，很简单地算出物体的 x 坐标与 y 坐标。而如果是速度中包含了加速度的圆周运动应当如何计算呢？比如系统中需要同时处理重力与空气阻力，或者处理不完整的圆周运动（例如游戏中角色使用绳索从一处荡到另一处）等。在这些情况下，如果还只是简单地对位置累加速度，对速度累加加速度，其结果肯定不是圆周运动。此时为了让物体仍然呈现圆周运动，我们要如何处理加速度呢？

这里需要先参考一下 1.4 节后半部分中的这两个等式

$$x = \int v\mathrm{d}t$$
$$v = \int a\mathrm{d}t$$

这是当施加的加速度确定时，求速度与位置的计算公式。圆周运动则正好相反，是需要根据确定的位置来算出加速度，所以只要把上面的等式逆运算就可以

了。而积分的逆运算，就是大家都知道的微分 。那么根据位置计算速度，根据速度计算加速度，就有以下等式成立。

$$v = \frac{\mathrm{d}x}{\mathrm{d}t}$$
$$a = \frac{\mathrm{d}v}{\mathrm{d}t}$$

使用这两个等式，就可以计算出圆周运动的加速度。首先仅考虑 x 方向，假设物体正在以原点为中心、以 r 为半径、以 ω 为角速度进行圆周运动，此时 x 方向旋转满足等式

$$x = r \cdot \cos(\omega t)$$

然后计算 x 方向的速度 v_x，

$$\begin{aligned} v_x &= \frac{\mathrm{d}x}{\mathrm{d}t} \\ &= \frac{\mathrm{d}}{\mathrm{d}t}\{r \cdot \cos(\omega t)\} \\ &= -r\omega \cdot \sin(\omega t) \end{aligned}$$

根据 v_x，就可以计算出 x 方向的加速度 a_x。

$$\begin{aligned} a_x &= \frac{\mathrm{d}v_x}{\mathrm{d}t} \\ &= \frac{\mathrm{d}}{\mathrm{d}t}\{-r\omega \cdot \sin(\omega t)\} \\ &= -r\omega^2 \cdot \cos(\omega t) \end{aligned}$$

上面的式子与原来的 x 坐标的计算公式

$$x = r \cdot \cos(\omega t)$$

相比，由于两边都含有 r 与 $\cos(\omega t)$，因此可以得到下面的关系等式。

$$a_x = -\omega^2 x$$

可以看出，为了以原点为中心做圆周运动，只需将当前的 x 坐标乘以 $-\omega^2$ 的结果作为 x 方 向的加速度即可。请注意 cos 及 sin 等三角函数在计算中已经被消去了。

接下来用同样的方式计算 y 坐标，有

$$
\begin{aligned}
y &= r \cdot \sin(\omega t) \\
v_y &= \frac{\mathrm{d}y}{\mathrm{d}t} \\
&= \frac{\mathrm{d}}{\mathrm{d}t}\{r \cdot \sin(\omega t)\} \\
&= r\omega \cdot \cos(\omega t) \\
a_y &= \frac{\mathrm{d}v_y}{\mathrm{d}t} \\
&= \frac{\mathrm{d}}{\mathrm{d}t}\{r\omega \cdot \cos(\omega t)\} \\
&= -r\omega^2 \cdot \sin(\omega t) \\
\therefore a_y &= -\omega^2 y
\end{aligned}
$$

同样可以看到，三角函数都被消去了。整理一下 a_x 、a_y 的结果，有

$$
\begin{cases}
a_x = -\omega^2 x \\
a_y = -\omega^2 y
\end{cases}
$$

从向量的角度来重新审视一下上面的等式，可以得出这样一个结论，即将物体所在位置的位置向量乘以 $-\omega^2$ 作为加速度，就会形成以原点为中心的圆周运动（参考图 1-6-4）。

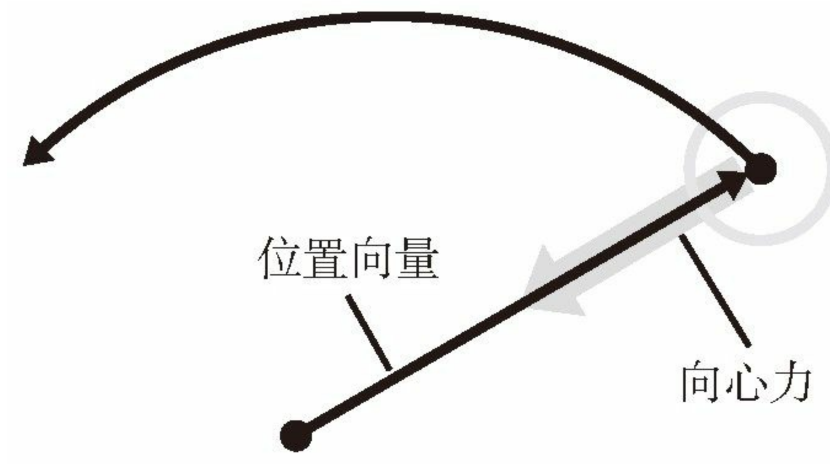


图 **1-6-4**　位置向量在加速度作用下进行圆周运动

等式中的负号，表示被施加的加速度（或力）是始终指向原点即旋转的中心方向的。于是我们就可以得出结论，即对物体施加一个指向某一点的角速度为 ω 的力，物体就会围绕该点做圆周运动。由于这个力始终指向旋转的中心，因此称为向心力 。使用向心力实际制作的圆周运动程序请参考示例程序 Movement_6_2.cpp。

请注意在这个程序中，InitCharacter 函数里决定物体的初始位置及初始速度的是以下部分。

代码清单 **1-6-2**　使用向心力的圆周运动中决定初始位置及初始速度的部分

```
rx = ROT_R;            // 初始位置
ry = 0.0f;
vx = 0.0f;             // 初始速度
vy = ROT_R * ANGLE_VEL;
```

可以看到初始位置为 (ROT_R, 0.0f)，即 (r , 0) 的位置，初始速度为 (0.0f, ROT_R * ANGLE_VEL)，即

$(0, r\omega)$。为什么要这样设置呢？这是因为如果想要围绕某个初始位置做圆 周运动，首先必须将物体放在圆周的某个点上才行。于是将 $t=0$ 代入刚才的公式

$$x = r \cdot \cos(\omega t)$$
$$y = r \cdot \sin(\omega t)$$

得到的 $(r, 0)$ 就是初始位置。其次，只要将 $t=0$ 代入对 x 、y 进行微分得到的关系式

$$v_x = -r\omega \cdot \sin(\omega t)$$
$$v_y = r\omega \cdot \cos(\omega t)$$

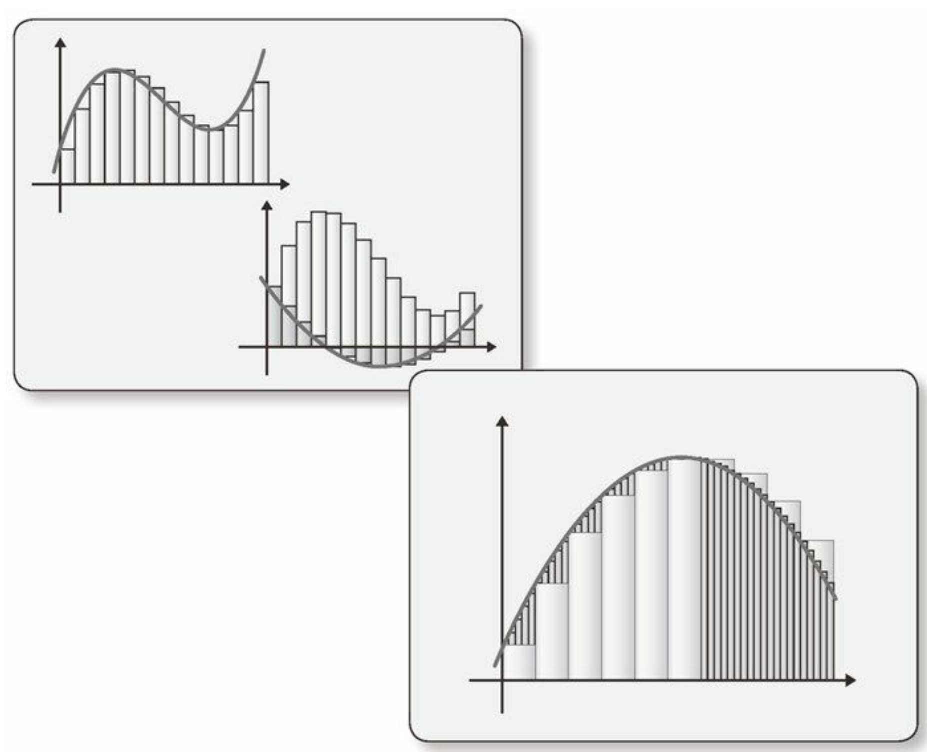
就可以得到初始速度 $(0, r\omega)$。

由于上面的算式中已经将三角函数全部消去了，因此这个程序中控制物体实际运动的 MoveCharacter 函数中并没有使用 sin、cos 等三角函数进行运算，只是简单地使用了加法乘法等，而最终效果与使用了三角函数的圆周运动是完全一样的。实际上 sin、cos 等运算比简单的四则运算要耗时很多，因此使用向心力实现的圆周运动，比使用三角函数实现的圆周运动在运算速度方面更胜一筹。

综上所述，我们掌握了两种在游戏中实现圆周运动的方法，一种基于三角函数，另一种则使用了向心力，大家可以根据具体情况选择使用恰当的方法。

1.7 [进阶]微分方程式及其数值解法

Key Word 微分方程、数值解法、欧拉法



在使用计算机表现物体运动时，运动是否正确是受条件限制的。在本小节，我们就来一起了解下这种限制存在的原因吧。

本章围绕着物体运动这一话题，编写了不少表现物体运动的程序实例。然而计算机终究只是一种在有限的

时间内进行计算的机器，它所能正确表现的物体运动自然是有限的。那么为什么会有这些制约条件呢？本小节就会对此进行说明。首先从基本原理开始，当一个物体被施加力时，该物体就会产生与所施加的力成比例的加速度。用公式表达就是

$$F = ma$$

F 代表物体被施加的力，m 是物体的质量，a 是物体产生的加速度。这就是有名的运动方程 （牛顿第二定律）。从这个公式中可以得到

$$a = \frac{F}{m}$$

而我们在 1.6 节中已经介绍过，位置、速度、加速度之间有以下关系。

$$v = \frac{\mathrm{d}x}{\mathrm{d}t}$$
$$a = \frac{\mathrm{d}v}{\mathrm{d}t}$$

进而得到

$$\begin{aligned} a &= \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right) \\ &= \frac{\mathrm{d}^2x}{\mathrm{d}t^2} \end{aligned}$$

等式中的 $\frac{\mathrm{d}^2x}{\mathrm{d}t^2}$ 代表将位置以时间进行 2 次微分。将其代入之前的运动方程式，可以得到

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = \frac{F}{m}$$

这种等式中含有微分的方程称为微分方程 。如果能将这个描述物体运动的微分方程求解，理论上来说就可以重现任何力作用下的物体运动。

然而要将这个看似简单的算式求解，也不是那么容易的。首先让我们来考虑最简单的情况。在这种情况下，只在微分方程两边对 t 进行 2 次积分就可以求解，比如 1.4 节中只对物体施加重力就属于这种情况。下面我们就来实际操作一下，给物体施加重力加速度 g ，即 mg 的力，这时有

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = \frac{mg}{m}$$
$$\therefore \frac{\mathrm{d}^2x}{\mathrm{d}t^2} = g$$

然后在等式两边对 t 进行积分，由于微分是积分的逆运算，所以左边的微分被消去一次，得到

$$\int \frac{\mathrm{d}^2x}{\mathrm{d}t^2}\mathrm{d}t = \int g \ \ \mathrm{d}t$$
$$\frac{\mathrm{d}x}{\mathrm{d}t} = gt + C_0$$

上式中，C_0 是左右两边的积分常数的和。对上式再次进行积分，得到

$$\int \frac{\mathrm{d}x}{\mathrm{d}t}\mathrm{d}t = \int gt + \int C_0$$
$$x = \frac{1}{2}gt^2 + C_0t + C_1$$

C_1 是两边的积分常数。这个式子与 1.4 节中我们推导出的等式一致。

但是在现实中几乎没有人会像这样对运动方程两边进行积分求解。例如，根据胡克定律，具有弹性的物体会被施加一个名为 $-kx$ 的力。k 称为劲度系数，弹簧越硬劲度系数越大。此时上式为

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2}=\frac{F}{m}$$
$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2}=\frac{-kx}{m}$$

这个式子就无法简单地对两边进行积分求解 x 。因为积分求解的对象是 x ，而 x 在等式右边也出现了，即使对两边进行积分，也仅仅是将等式从微分变成了积分，没有什么实际意义。下面来实际操作一下。

$$\int\frac{\mathrm{d}^2x}{\mathrm{d}t^2}\mathrm{d}t=\int\frac{-kx}{m}\mathrm{d}t$$
$$\frac{\mathrm{d}x}{\mathrm{d}t}=-\frac{k}{m}\int x\ \ \mathrm{d}t+C_0$$
$$\int\frac{\mathrm{d}x}{\mathrm{d}t}\mathrm{d}t=\int\left(-\frac{k}{m}\int x\ \ \mathrm{d}t+C_0\right)\mathrm{d}t$$
$$x=-\frac{k}{m}\int\left(\int x\ \ \mathrm{d}t\right)\mathrm{d}t+C_0t+C_1$$

求得的结果中仍然包含 x ，无法解决问题。针对上式，最简单的解决方式是换个角度，既然求解 2 次微分的目的是消去未知数的微分或积分符号，那么如果有一个函数通过 2 次积分后能保持不变就可以解决这个问题了。比如 sin 函数就有这样的特性，sin 函数进行 1 次微分后会变成 cos 函数，再次进行微分又会变回 sin 函数。那么就让我们按这个思路来试试这个方法是否可行吧。首先，假设

$$x=A\cdot\sin(\omega t)$$

这里的 A 与 ω 都是常数。然后对等式进行微分。顺便提一下，下面的求解过程与 1.6 节中对圆周运动的求解过程几乎是一样的。

$$
\begin{aligned}
x &= A \cdot \sin(\omega t) \\
\frac{\mathrm{d}x}{\mathrm{d}t} &= A\omega \cdot \cos(\omega t) \\
\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} &= -A\omega^2 \cdot \sin(\omega t) \\
\therefore \frac{\mathrm{d}^2 x}{\mathrm{d}t^2} &= -\omega^2 x
\end{aligned}
$$

对比一下具有弹力的物体的运动方程

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = \frac{-kx}{m}$$

可以看到，如果 $\omega^2 = \frac{k}{m}$ ，则两个方程是一致的，所以 $x = A \cdot \sin(\omega t)$是满足此运动方程的。即 $\omega = \pm\sqrt{\frac{k}{m}}$ ，此时方程的解为

$$x = A \cdot \sin\left(\pm\sqrt{\frac{k}{m}} \cdot t\right)$$

如果对这个答案有怀疑，可以将解代入运动方程实际验证一下是否真的成立。不过比起之前求解用的普通方程，微分方程的求解过程总是让人感觉不够工整，就连最终求出的解也像是东拼西凑出来的。

上面推导出了满足带弹簧的物体的运动方程的解，但是需要注意的是这个运动方程并不是只有这一个解。我们所求出的 $x = A \cdot \left(\pm\sqrt{\frac{k}{m}} \cdot t\right)$这个等式只是满足该运动方程的一个解，并不能否定其他解存在的可能性。实际上也可能会得到其他形式的解，本书中不再深入谈论。

上文中我们介绍了运动方程的两种解法：对等式两边积分以及利用三角函数。那么所有的运动方程是不是都可以通过这两种方法求解呢？很遗憾这是做不到

的。微分方程可以分为线性微分方程与非线性微分方程两种。上面的例子中，重力的等式与弹簧的等式都是线性微分方程。数学家研究得出的结论是，线性微分方程是必然有解的，而非线性微分方程会存在一些无论如何都无法求解的情况。

举一个游戏开发者身边的例子，这个例子涉及流体力学的情况。比如挥动一面旗帜，这时的运动就不能通过简单的函数来表示。在这些情况下，我们就需要使用微分方程的数值解法，即牺牲了一定程度的正确性的数值计算。实际上 1.4 节中最开始的程序以及本章的很多问题都已经在使用这种方法了，只不过之前的所有算法，都仅仅采用了数值解法中最简单的、误差最大的算法。具体来说，就是在欧拉法中将 Δt 置为 1 的解法。下面将进一步对欧拉法进行说明。

所谓欧拉法，就是通过逐步计算来求得微分方程的近似解。这里的“逐步”具体来说有点类似程序中的循环，根据前一次的计算结果来进行下一次计算。在求解微分方程时，如果想要得到最精确的结果，采用欧拉法只能永无止境地计算下去，因为每次计算都只能得到当前的结果。因此对于复杂的非线性微分方程，想要精确求解本来就是不可能的，我们只能努力求得某个可以接受的精度的解。现实中使用欧拉法时，虽然可以进行多次逐步求解，但往往只会求第一次近似这种最简单的近似解，也就是将微分方程近似为差分方程来求解。具体来说就是以下的近似过程。

首先，位置 x 与速度 v 之间本来就有

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v$$

这样的关系。这个等式的左边是微分，如果想要通过速度计算位置，原本是必须进行积分的。但是在欧拉法中可以近似为

$$\frac{\Delta x}{\Delta t} = v$$

这里的 Δt 是时间间隔（在游戏中就是 1 帧 = $\frac{1}{60}$ 秒），Δx 是在 Δt 时间内位置 x 的变化量。也就是说，Δx 其实就是现在的位置减去前一次的位置。假设将现在的位置表示为 x_n，前一次的位置表示为 x_{n-1}，上面的等式就可以表示为

$$\frac{x_n - x_{n-1}}{\Delta t} = v$$

即

$$x_n = x_{n-1} + v\Delta t$$

同理，速度与加速度的关系为

$$v_n = v_{n-1} + a\Delta t$$

将两个式子并列起来得到联合方程组

$$\begin{cases} x_n = x_{n-1} + v\Delta t \\ v_n = v_{n-1} + a\Delta t \end{cases}$$

这就是欧拉法的等式。并且这里第一个等式中的 v 可以直接使用第二个等式中的 v_n 或 v_{n-1}。

在上面使用过的近似等式

$$\frac{\Delta x}{\Delta t} = v$$

中，当 Δt 取无限小时，左边就直接转化为了微分，也就不是近似了。用等式可以书写为

$$\lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} = \frac{\mathrm{d}x}{\mathrm{d}t}$$

也就是说，欧拉法的等式

$$\begin{cases} x_n = x_{n-1} + v\Delta t \\ v_n = v_{n-1} + a\Delta t \end{cases}$$

在使用第一次近似时可能会采用较粗略的计算，但其实如果 Δt 足够小的话，其计算精度也会上升。而在本章的 1.4 节中也提到过，游戏中经常会把欧拉法的等式中的 Δt 设置为 1，即

$$\begin{cases} x_n = x_{n-1} + v \\ v_n = v_{n-1} + a \end{cases}$$

这样计算所使用的时间间隔就变为了 1 帧，即 1/60 秒。在物体被施加了加速度的情况下，这样计算所产生的误差是无法被忽视的。即便如此，很多游戏中也并不会去要求更高的计算精度，因为此时物体的动作肉眼观察起来并不会感到奇怪，这就足够了。但如果是球类游戏等对物体运动的正确性要求很高的话，上文采用的 $\Delta t = 1$ 就显得精度不足了。此时为了提高精度，一般有以下三个方法。

① 使用精度更高的高阶近似求解微分方程，比如龙格 - 库塔法（Runge-Kutta methods）等。

② 使用线性多步法（Linear multistep method）计算，即不光使用前一次的值，还使用前两次或更早的值进行计算，比如 Adams-Bashforth 法等。

③ 在一帧内多次使用欧拉法计算，缩小Δt 。

方法①是偏重理论、数学层面的方法。无论是使用龙格 - 库塔法或者更高阶的近似（如 4 阶），在同样的时间间隔 Δt 内都能得到比欧拉法更好的精度，这也是在模拟实验时常使用的方法。但是实际上在使用龙格 - 库塔法时，代码的可维护性会下降，因为龙格 - 库塔法在理论上要比欧拉法复杂很多，如果不是高校物理专业的人可能很难运用自如。而龙格 - 库塔法同时还存在很多版本的算法，可能会造成混乱。因此如

果代码需要让自己以外的人（比如对物理不够熟悉的人）维护，考虑到代码的可维护性，应当慎重考虑是否真的需要采用龙格 - 库塔法。

方法②的 Adams-Bashforth 法理论上比龙格 - 库塔法略简单一些，也没有那么多不同版本的算法，代码可维护性会好一些。但是毫无疑问还是要比欧拉法复杂，在使用时也需要斟酌。

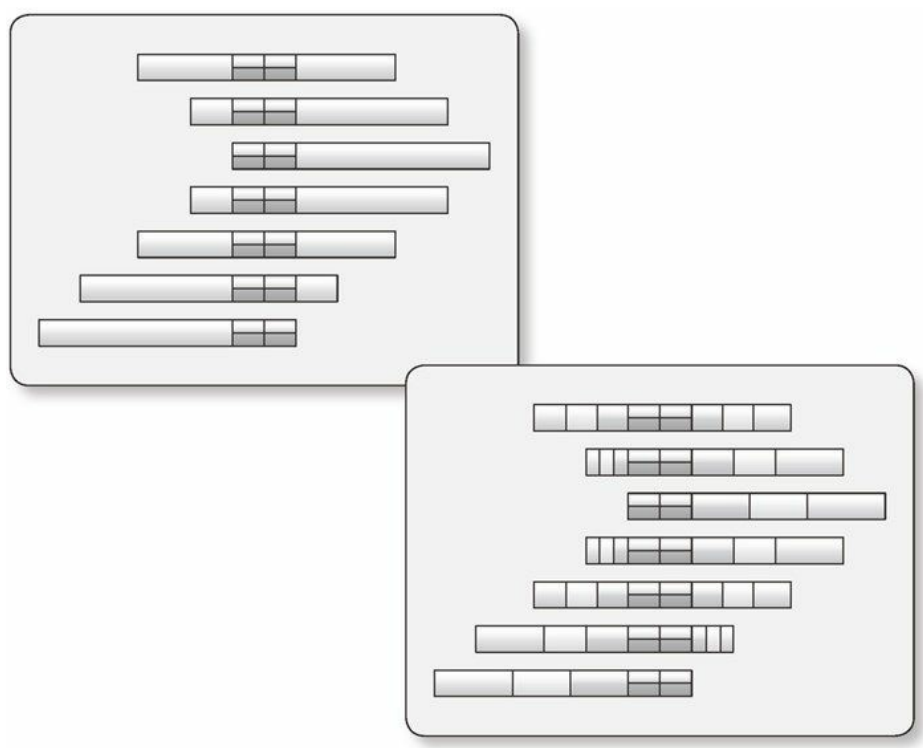
方法③采用的是容易理解的欧拉法，从程序的可维护性上来说应该是最好的方法。比如将欧拉法等式的 Δt 设为 0.1，进行 10 次循环，就可以很简单地得到高精度的运算结果，而在时间精度提高的同时，其他物体的运动是否正确也变得更加容易判断。虽然这样做可能会花费更多的计算时间，但是现在的计算机性能都非常高，游戏中比起物体运动的计算等，渲染其实更加消耗时间，因此个人认为如果游戏中需要高精度的物理计算，不要去使用龙格 - 库塔法或 者 Adams-Bashforth 法，重复使用欧拉法就是最好的方法。因此这里不再列举龙格 - 库塔法及 Adams-Bashforth 法的详细公式，对这些高级方法有兴趣的读者，不妨自己去深入研究一下。

第 2 章　卷动

2.1 将背景从一端卷动到另一端

Key Word 镜头位置、卷动幅度、比例关系

RANK/easy 前半部分　RANK/normal 后半部分



早期的游戏基本上只有一个画面，而现在的游戏中用一个以上的画面作为背景已经成为标配。本小节就让我们一起学习如何制作基本的背景卷动吧。

本小节将讲解如何将背景图片从一端卷动到另一端。为了便于说明，这里只使用一张插画作为背景，而不使用背景卷动中常见的地图块（map chip）方法（即将一张大背景图拆分成若干小块），这也是很多 2D

游戏尤其是格斗游戏中经常使用的方法。

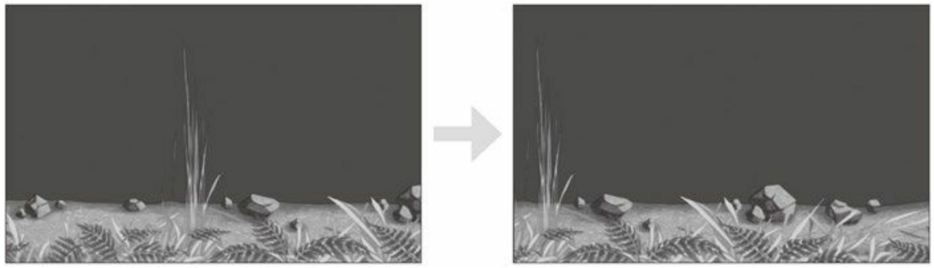


图 **2-1-1**　背景从一端卷动到另一端的程序

示例程序 Scroll_1_1.cpp 就是一个将图片从一端卷动到另一端的程序。代码有点长，其中最重要的是以下部分（代码清单 2-1-1）。

代码清单 **2-1-1**　背景卷动处理的主要部分（**Scroll_1_1.cpp** 片段）

```
int InitBack( void )                                    /
{
    fCamera_x = VIEW_WIDTH / 2.0f;                      /
    fBack_x = VIEW_WIDTH / 2.0f - fCamera_x;            /

    return 0;
}


int MoveBack( void )                                    /
{
    // 按左方向键时向左卷动
    if ( GetAsyncKeyState( VK_LEFT ) ) {
        fCamera_x -= CAMERA_VEL;
        if ( fCamera_x < VIEW_WIDTH / 2.0f ) {
            fCamera_x = VIEW_WIDTH / 2.0f;
        }
    }
    // 按右方向键时向右卷动
    if ( GetAsyncKeyState( VK_RIGHT ) ) {
        fCamera_x += CAMERA_VEL;
        if ( fCamera_x > ( float )( PICTURE_WIDTH
            fCamera_x = ( float )( PICTURE_WIDTH
        }
    }
```

```

    fBack_x = VIEW_WIDTH / 2.0f - fCamera_x; //

    return 0;
}
```

其中，InitBack 函数设置了镜头（视点）的位置（即变量 fCamera_x）及背景图片的显示位置（即变量 fBack_x）。变量 fCamera_x 控制的是背景图片的哪一部分正在被看到（即图片的 *x* 坐标）。上面程序中将视线焦点设置为了画面中心。fBack_x 控制的是绘制背景图片左端的 *x* 坐标。因此在 InitBack 函数中，fCamera_x 就为画面宽度的一半的坐标，即 VIEW_WIDTH/2.0 f 的位置（参考图 2-1-2）。

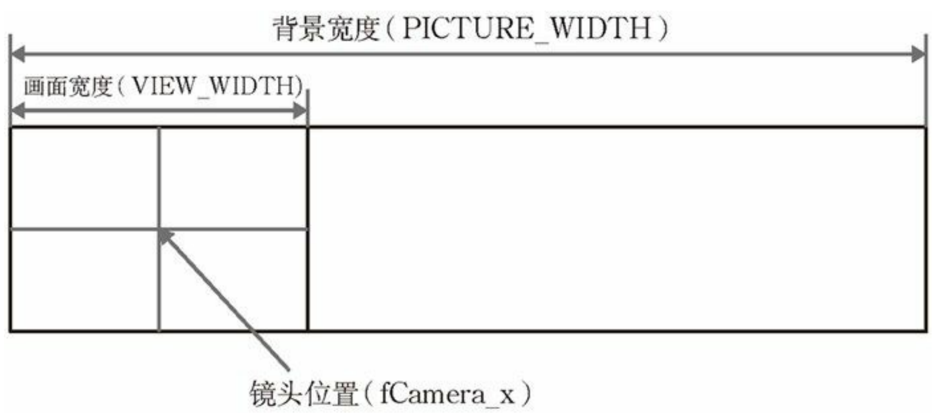


图 **2-1-2**　镜头位置与背景显示部分的关系

fBack_x 则是根据 fCamera_x 得到的。

```
    fBack_x = VIEW_WIDTH / 2.0f - fCamera_x;    /
```

请注意镜头和背景图片是反方向运动的，镜头左移图片就会向右卷动，镜头右移图片则向左卷动。因此程序中便通过 ××× - fCamera_x 这种形式，让背景可以按镜头坐标的反方向运动。当背景图片的 *x* 坐标（fBack_x）为 0 时，镜头正好在画面的中央，此时 *x* 坐标的位置为 VIEW_WIDTH / 2.0 f，背景图片在 VIEW_WIDTH / 2.0 f - fCamera_x 的位置。所以 fBack_x 的初始值为 VIEW_WIDTH / 2.0 f - VIEW_WIDTH / 2.0 f = （0 参考图 2-1-2）。也就是说，卷动是从最左端开始的。

在每帧调用的函数 MoveBack 中，根据键盘输入移动背景。首先来看按左方向键时的情况。

```
    // 按左方向键时向左卷动
    if ( GetAsyncKeyState( VK_LEFT ) ) {
        fCamera_x -= CAMERA_VEL;
        if ( fCamera_x < VIEW_WIDTH / 2.0f ) {
            fCamera_x = VIEW_WIDTH / 2.0f;
        }
    }
```

代码控制镜头以 CAMERA_VEL 的速度向左运动，同时还对移动范围作了一定的限制。代表镜头的 *x* 坐标的 fCamera_x 移动到 VIEW_WIDTH / 2.0 f 即画面宽度一半的位置时就无法再向左移动了，因为镜头的 *x* 坐标移动到画面宽度一半的位置时，背景的左端也正好到达画面的左端（参考图 2-1-2）。这与 InitBack 函数中 fCamera_x 与 fBack_x 之间有差值是同样的道理。

当按右方向键时，MoveBack 函数的处理如下。

```
    // 按右方向键时向右卷动
    if ( GetAsyncKeyState( VK_RIGHT ) ) {
```

```
            fCamera_x += CAMERA_VEL;
            if ( fCamera_x > ( float )( PICTURE_WIDTH
                fCamera_x = ( float )( PICTURE_WIDTH
            }
        }
```

镜头将以 CAMERA_VEL 的速度向右移动。与向左时同样，这里也对移动范围进行了限制。向右时镜头最多能移动到背景图的宽度（PICTURE_WIDTH）减去画面宽度的一半（VIEW_WIDTH / 2.0 f）的位置。即画面的右端与背景图右端重叠时就是向右卷动的极限位置（参考图2-1-3）。

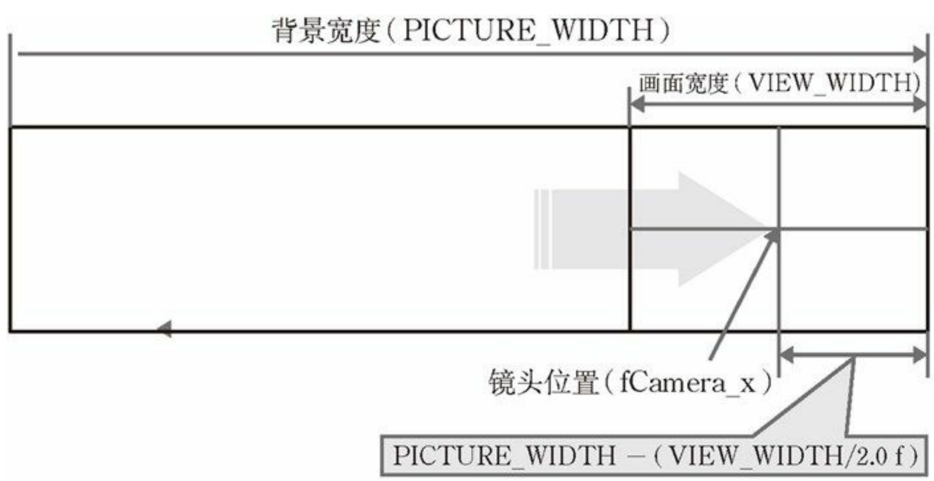


图 **2-1-3**　画面到达背景的右端

至此我们可以得出结论：背景图片的左边和右边都存在一块镜头无法进入的区域，其面积为画面宽度的一半。这是由于画面最多只能显示宽度为 VIEW_WIDTH 的图像，所以画面也只能在不超出背景图片的范围内移动。而镜头移动的范围正好比背景图片的宽度少一个画面那么宽（具体为 VIEW_WIDTH / 2.0 f+ 向右的 VIEW_WIDTH / 2.0 f），因此如果背景图片正好与画面一样宽，则将无

法卷动（参考图 2-1-4）。

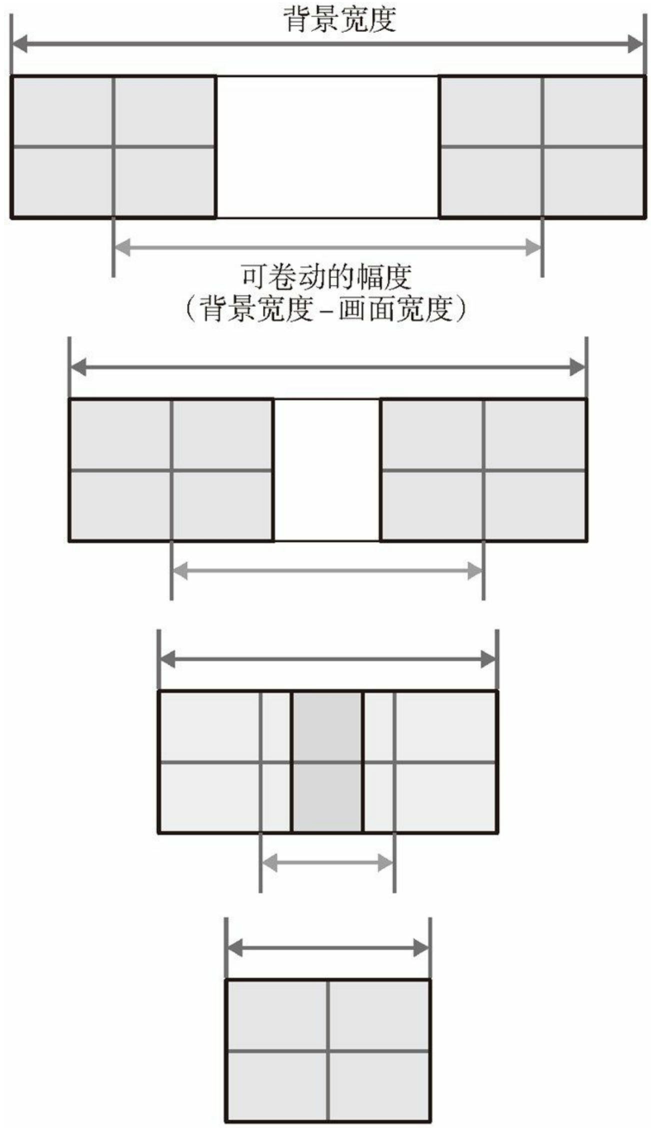


图 **2-1-4**　当背景宽度与画面宽度一样时将无法卷动

最后一行代码

```
fBack_x = VIEW_WIDTH / 2.0f - fCamera_x; //
```

是通过镜头的 *x* 坐标 fCamera_x 计算背景的 *x* 坐标 fBack_x，计算方法与 InitBack 函数中计算 fBack_x 的方法一样。

POINT　像上文这样同一个计算过程在多个地方使用时，一般都会将其转化为函数，以避免出现多处重复代码。否则，当之后进行调试或遇到需求变更等情况时，就要一处一处地修改代码，这样不仅会导致程序的可维护性变差，也更容易出错。但是，由于本次所用到的计算都非常简短，因此与其写成函数放在别处，不如直接写出算式更加具备可读性。于是，上例的代码在同时考量了代码的可维护性与可读性后，优先照顾了代码的可读性，在两个地方写了同样的计算语句。

- 多重卷动的实现

我们已经实现了从一端到另一端的背景卷动，为了让卷动更加具备立体感，接下来让我们来实现多重卷动 。



图 **2-1-5**　三重卷动程序

示例程序 Scroll_1_2.cpp 实现了 3 个图片重叠的卷动效果，即三重卷动程序。程序中最重要的部分是 MoveBack 函数中的以下 3 行（代码清单 2-1-2）。

代码清单 **2-1-2**　多重卷动处理的主要部分（**Scroll_1_2.cpp** 片段）

```
    fBack_x1 = VIEW_WIDTH / 2.0f - fCamera_x
    fBack_x2 = ( float )( PICTURE_WIDTH2 - V
    fBack_x3 = ( float )( PICTURE_WIDTH3 - V
```

其中 fBack_x1、fBack_x2、fBack_x3 分别代表外侧的图片（图 1）、中间的图片（图 2）、里侧的图片（图 3）的 *x* 坐标。可以看出，与之前只有一张背景图的卷动程序 Scroll_1_1.cpp 相比，有以下一些不同。

- 外侧图片的卷动处理与只有一张背景图的程序相同。
- 中间图片的 *x* 坐标为外侧图片的 *x* 坐标乘以 (PICTURE_WIDTH2 - VIEW_WIDTH) / (PICTURE_WIDTH1 - VIEW_WIDTH)。
- 里侧图片的 *x* 坐标为外侧图片的 *x* 坐标乘以 (PICTURE_WIDTH3 - VIEW_WIDTH) / (PICTURE_WIDTH1 - VIEW_WIDTH)。

可以看到这个程序是以外侧图片的卷动为基准的，其他图片的运动本质上都是与外侧图片成一定速度比例的联动。那么如何决定这些图片卷动的速度比例呢？如果从最终呈现的视觉效果来考虑，这些图片卷动的速度比例，应该是由他们在视觉上体现的纵深程度来决定的。但是在示例程序 Scroll_1_2.cpp 中，并没有根据视觉的纵深程度去编写程序，而是根据每张图片的宽度来决定其卷动速度的。图片的宽度越大卷动速度越快，图片宽度越小，对应的卷动速度也会降低。因为在游戏开发中，背景卷动的速度设定一般取决于

游戏策划或设计师，但是他们却很难给出一个具体的数据。这种情况下，通过按示例程序这样处理，程序开发就只需要拿到背景图片就可以开工了，从而不仅提升了开发效率，也不容易出BUG。

为了简化说明，示例程序 Scroll_1_2.cpp 中将每张图片的宽度用 define 进行了硬编码，但在正式项目中，建议通过程序读取图片的宽度，这样当图片的宽度更改时，卷动速度也会自动做出相应的更改，不必每次都去修改程序。

> **POINT** 卷动速度的计算应该参考背景图片的宽度，这样可以提高程序员与设计师之间的沟通效率。

既然程序是根据图片的宽度来决定卷动速度的，那么具体怎样通过图片宽度计算出卷动的速度比例呢？首先我们已经想到，图片越宽卷动越快，图片越窄卷动越慢，那么根据图片宽度的比例调整卷动速度的比例是不是就可以了呢？假设基准图片的宽度为 w_1，联动的图片宽度 为 w_2，基准图片的位置为 x_1，联动的图片位置为 x_2，根据比例计算有

$$x_1 : x_2 = w_1 : w_2$$
$$\therefore x_2 = \frac{w_2}{w_1} x_1$$

显然是不正确的。举一个极端的例子，比如联动的图片宽度 w_2 正好与画面（指玩家看到的画面）宽度相等会怎样呢？此时按照上文的结论，图片应该是无法卷动的。但是在上面这个比例关系式中，图片仍然会以非零的速度卷动，这显然是有问题的。

为了正确计算卷动速度，不能只选择图片的宽度，而应该同时使用图片宽度与画面宽度进行计

算，从而得到卷动的最大范围。如果仅从卷动本身来考虑的话，卷动的最大范围应该是图片宽度减去画面宽度所得到的值（参考图 2-1-4）。那么如果画面宽度为 w_V，基准图片的卷动范围则为

$$w_1 - w_V$$

联动图片的卷动范围为

$$w_2 - w_V$$

此时只要通过计算卷动范围与速度的比例

$$x_2 = \frac{w_2 - w_V}{w_1 - w_V} x_1$$

就可以由基准图片的位置计算出联动图片的位置。上面的等式中，如果图片的宽度 w_2 与画面的宽度 w_V 相等，系数的分子 $w_2 - w_V$ 为 0，将无法卷动，符合我们的预期。所以示例程序 Scroll_1_2.cpp 中使用了以下算式。

```
    fBack_x2 = ( float )( PICTURE_WIDTH2 - V
    fBack_x3 = ( float )( PICTURE_WIDTH3 - V
```

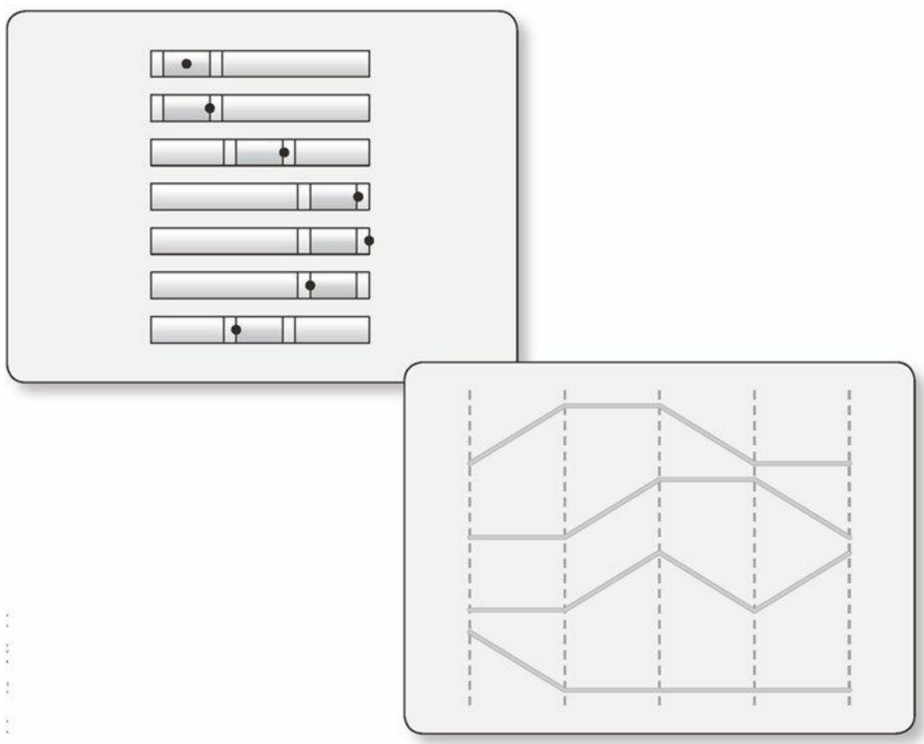
可见，卷动的基准始终是图 1，图 2、图 3 的显示位置都通过上述等式计算得到。

当画面上有多个不同速度的图片卷动时，以其中一个图片为基准，并基于基准图片来控制其他图片会非常方便。

2.2 让背景卷动与角色的运动产生联动

Key Word 区域坐标、画面坐标

RANK/normal



比起单纯的背景卷动，如果背景能与游戏角色的动作产生联动，效果会更好。在本小节，我们就来学习能与角色的动作产生联动的背景卷动处理。

本小节将讲解能与玩家控制的角色（下文简称角色）产生联动的背景卷动处理。在游戏中，背景卷动大致

有两种：一种是像射击游戏那样的强制的背景卷动，卷动与角色的动作基本没有关系；另一种更加普遍，比如在动作游戏或格斗游戏中，卷动根据角色的运动进行联动。



图 **2-2-1**　与角色运动联动的背景卷动程序

与角色运动联动的背景卷动程序如 Scroll_2_1.cpp 所示。在这个程序中，角色向右移动时背景向左卷动，角色向左移动时背景向右卷动，据此来尽可能地使角色定位在画面中央。也就是说，对角色的操作并不会反映在角色本身，而是表现为背景的卷动（参考图 2-2-2）。这与上一小节的处理是差不多的（只是镜头视点的位置没有角色而已）。

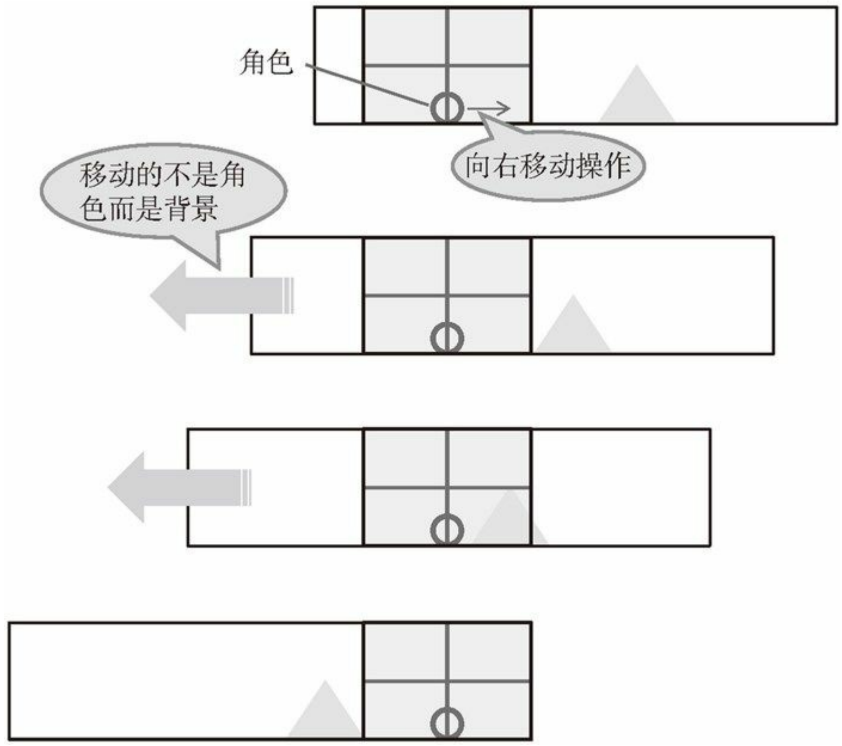


图 **2-2-2**　根据角色操作只移动背景

但是这样处理就无法使角色移动到背景（这里也可以称作区域[1]）的两端。上一节中已经讲过，这是由于背景的卷动范围是有边界的。而我们可以通过一些处理，当角色移动到超出背景可以卷动的范围时停止背景卷动，使角色继续相对画面移动并碰到两端（参考图 2-2-3）。

[1] 本章中所谓的区域是指背景图片覆盖的范围，即角色能移动的范围。

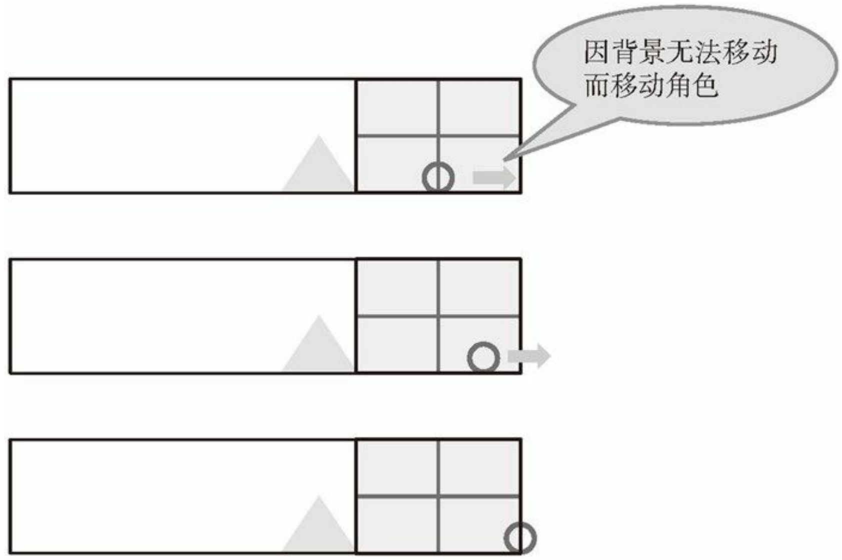


图 **2-2-3**　背景到达画面两端后只移动角色

实现以上效果的关键代码请参考 Scroll_2_1.cpp 中 MoveChara 函数的以下部分（代码清单 2-2-1）。

代码清单 **2-2-1**　只在画面两端移动角色（**Scroll_2_1.cpp** 片段）

```
fCamera_x = fChara_sx;        // 镜头暂时回到
if ( fCamera_x < VIEW_WIDTH / 2.0f ) { // 检
    fCamera_x = VIEW_WIDTH / 2.0f;
}
if ( fCamera_x > PICTURE_WIDTH - VIEW_WIDTH /
    fCamera_x = PICTURE_WIDTH - VIEW_WIDTH /
}
fChara_x = fChara_sx - fCamera_x + VIEW_WIDT
fBack_x = VIEW_WIDTH / 2.0f - fCamera_x;
```

以下是代码中涉及的坐标：

- fCamera_x 为区域中镜头的 *x* 坐标

- fChara_sx 为区域中角色的 *x* 坐标
- fChara_x 为画面上角色的 *x* 坐标
- fBack_x 为画面上背景的 *x* 坐标

这里同时出现了描述角色移动范围的区域坐标与画面坐标两个概念，请注意不要混淆。按代码的逻辑，角色首先会在区域内移动，此时与镜头没有关系，然后镜头才会追踪角色的运动。因此画面上角色的坐标是通过角色与镜头在区域内的坐标求出来的。

为了使镜头追踪角色，代码做了以下处理。

```
    fCamera_x = fChara_sx;        // 镜头暂时回到
```

这是让镜头无条件地保持与角色处于同一位置，即此时角色与镜头正好同在画面的中央。但是如前所述，镜头比角色有着更加严格的移动限制，因此不可能让角色始终停留在画面中央。当镜头偏离角色的位置时，会通过以下方法检查镜头是否超过了左方向的边界。

```
    if ( fCamera_x < VIEW_WIDTH / 2.0f ) { // 检
        fCamera_x = VIEW_WIDTH / 2.0f;
    }
```

为了让背景不超出画面，必须使镜头到达从背景图片左端起到画面宽度的一半位置时就不再移动（参考图2-1-2），当镜头超过该边界时就将其重置回该位置。

同理也会对镜头是否超出右边界做如下检查，并且在必要时修正镜头位置。

```
    if ( fCamera_x > PICTURE_WIDTH - VIEW_WIDTH /
        fCamera_x = PICTURE_WIDTH - VIEW_WIDTH /
    }
```

经由上述处理，镜头就会尽可能地保持角色在画面中央，实在无法实现时，则将角色尽量显示在靠近中央的位置。

一旦确定了角色位置与镜头位置，就可以计算出角色在画面上的位置了。这个计算实际上是代码的以下部分。

```
    fChara_x = fChara_sx - fCamera_x + VIEW_WIDTH
```

首先能决定角色在画面上的位置的是 fChara_sx - fCamera_x 这一部分。这个值表示从镜头看到的角色的相对位置，当镜头正好与角色位置一致时，这个值为 0。将这个相对位置换算为实际的画面上的坐标时，就要利用镜头与角色位置重合时角色正好位于画面中央这一特性。由于镜头与角色重合时 fChara_sx - fCamera_x 为 0，因此通过加上画面宽度的一半 VIEW_WIDTH / 2.0 f，就可以得到画面中央的坐标了。这样一来就得到了角色在画面上的坐标，而由于角色自身也有一定的大小，角色坐标是从角色图片的左上角开始计算的，所以为了将角色正确地显示在画面中央，还要减去角色本身宽度的一半，即 CHARA_WIDTH / 2.0 f。最终结果就是角色在画面上

的 *x* 坐标，即上面程序中的等式。

- 只在角色靠近画面两端时才进行背景卷动的联动

接下来让我们通过与角色动作的联动来实现一个更逼真的卷动效果。在 Scroll_2_1.cpp 中，只要没有超出可以卷动的边界，角色就始终保持在画面中央，为了实现这种效果，我们移动了镜头。但是为了让角色保持在画面中央，角色稍微向左右移动一下，背景就要跟着卷动，这会让人觉得画面很不稳定，看起来会有点难受。于是我们可以这样改进：让角色在画面中央附近移动时背景不随着卷动，只有当角色靠近画面两端时背景才开始卷动（参考图 2-2-4）。

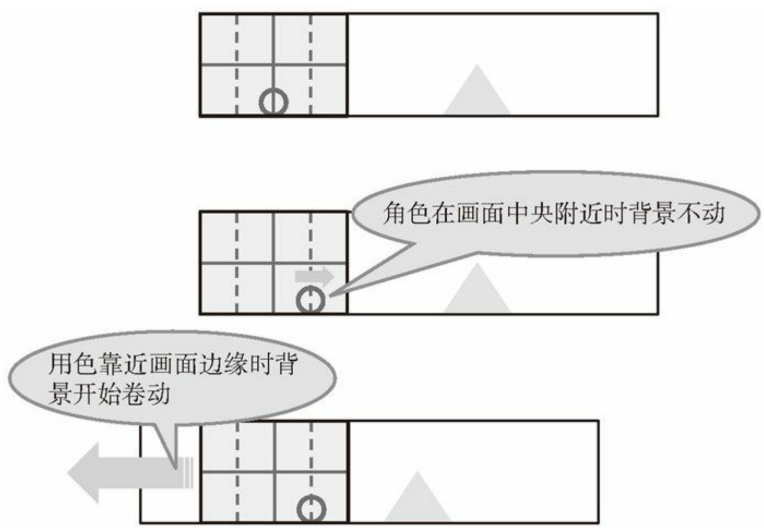


图 **2-2-4**　角色靠近画面两端时背景开始卷动

示例程序 Scroll_2_1a.cpp 实现了上述逻辑，其中重要的是 MoveChara 函数中的以下部分（代码清单 2-2-2）。

代码清单 **2-2-2**　只有满足条件时才移动镜头（**Scroll_2_1a.cpp**）

```
050  |     if ( fCamera_x < fChara_sx - SCROLL_DIF
```

```
            fCamera_x = fChara_sx - SCROLL_DIF;
        }
        if ( fCamera_x > fChara_sx + SCROLL_DIF
            fCamera_x = fChara_sx + SCROLL_DIF;
        }
```

与 Scroll_2_1.cpp 不同的是，镜头不再每次都被强制移动了。按照代码清单 2-2-2 的程序，只有当满足条件时镜头才会移动。下面就来具体分析一下。最开始的 if 语句中有以下内容：

```
        if ( fCamera_x < fChara_sx - SCROLL_DIF
            fCamera_x = fChara_sx - SCROLL_DIF;
        }
```

其中 fCamera_x 是区域中镜头的 *x* 坐标，fChara_sx 是区域中角色的 *x* 坐标。当角色到画面边缘的间距变小时，背景开始卷动，而事实上角色到画面边缘的间距变小，也就等价于角色到画面中央的间距变大（参考图 2-2-5）。

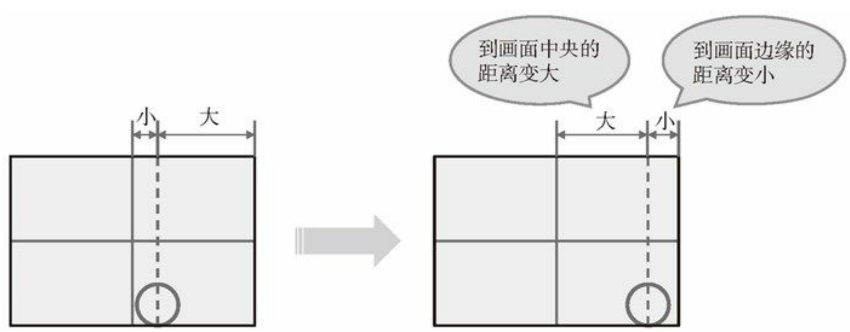


图 **2-2-5**　当角色到画面边缘的距离变小时，到画面中央的距离会变大

而画面中央到角色的间距，是由区域内镜头位置与角色位置的差决定的。也就是说，当镜头位置与角色位置的差为零时，角色就正好位于画面中央。于是只要检查镜头相对于角色是否靠近左侧，当靠近时就强制移动镜头即可。此时运行语句

```
    if ( fCamera_x < fChara_sx - SCROLL_DIF
```

检查镜头相对角色偏左的距离是否超过了 SCROLL_DIF 像素。如果镜头与角色的距离没有超过常数 SCROLL_DIF，镜头是不需要移动的。而镜头不移动，就意味着角色在画面中央附近，此时背景也不卷动。

下面为第二个 if 语句。

```
    if ( fCamera_x > fChara_sx + SCROLL_DIF
        fCamera_x = fChara_sx + SCROLL_DIF;
    }
```

同理，检测镜头是否在角色右侧，当偏右超过一定值时开始移动镜头。其中同样通过

```
    if ( fCamera_x > fChara_sx + SCROLL_DIF
```

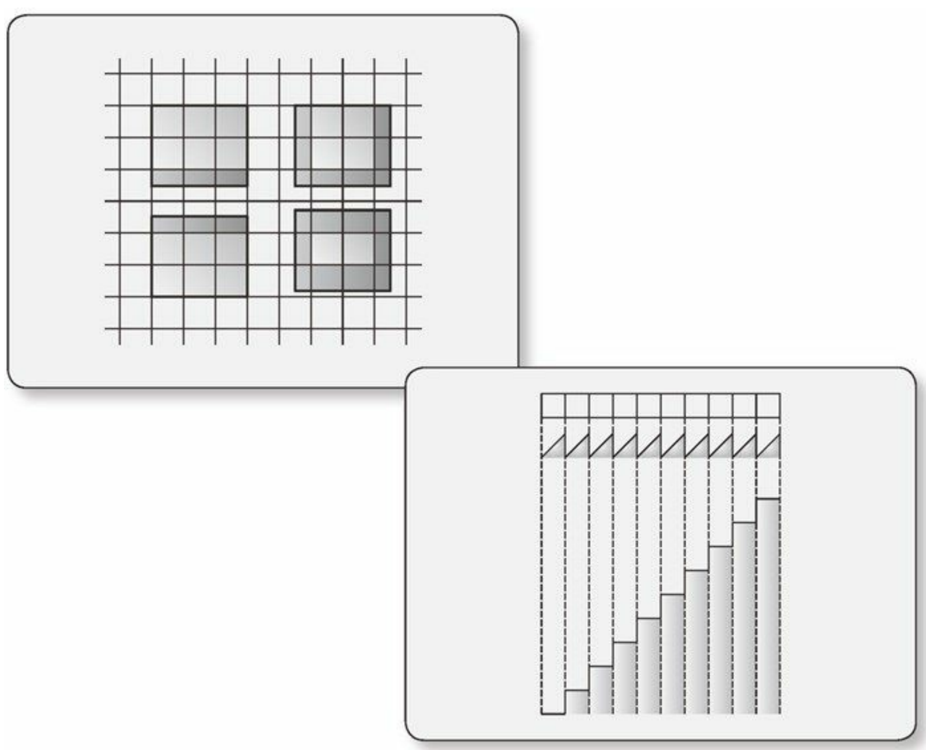
检查镜头相对角色偏右的距离是否超过了SCROLL_DIF，当大于 SCROLL_DIF 时，镜头就会移动到角色偏右 SCROLL_DIF 处的位置。

在上面的两个 if 语句之后，MoveChara 函数内还有防止角色移动超出背景可移动范围的处理，以及通过镜头位置计算角色与背景位置的处理，这部分与 Scroll_2_1.cpp 是一致的，如果对这部分处理有疑问，可以参考本小节开头对Scroll_2_1.cpp 的讲解。

2.3 卷动由地图块组合的地图

Key Word 地图、地图块、整数的减法、移位运算、逻辑运算

RANK/normal



制作大尺寸的背景不单耗时耗力，还会因为文件过大而对计算机造成高负荷。本小节就让我们来一起学习通过组合地图块的方式来实现大尺寸的背景图。

下面我们就来看看如何通过组合地图块来显示大地图（区域或背景），并实现卷动。

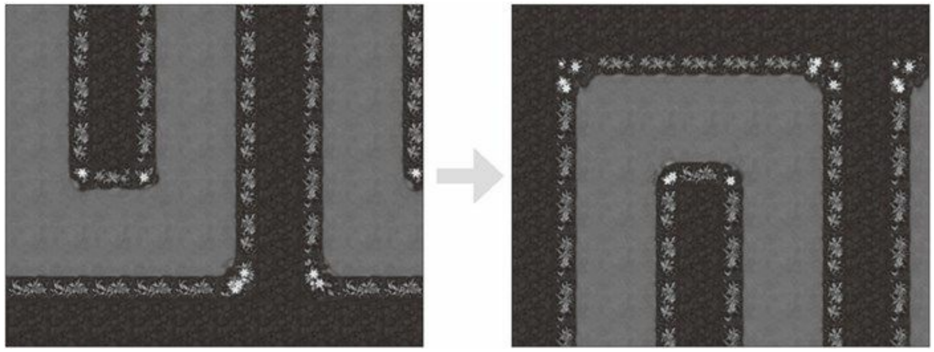


图 **2-3-1**　地图块组合的地图卷动

游戏中经常需要展示一张尺寸很大的区域或背景，而将背景全部放在一张图中显然是不现实的。比如当画面大小为 640×480 像素时，如果要展示一张该画面长度 ×10、宽度 ×10 的区域并实现卷动会怎么样呢？大家不妨来计算一下，此时的地图为 6400×4800 像素，用真色彩显示的话，1 像素就需要 32bit（=4 字节），6400 × 4800 × 4 = 122880000 字节，也就是说，仅背景图片就会消耗 122MB 内存之多。如果考虑到 2D 游戏中常用的多重卷动、分屏画面等，还会消耗更多的内存。

虽然目前主流的电视游戏机以及电脑都装载了 122MB 以上的内存，但是考虑到更高的分辨率、其他素材的容量、加载时间、掌上游戏机等因素，依然使用这个级别的数据显然是不现实的。因此常用的解决方法是将一个大地图分解为若干个小的“地图块”，1 个地图块还可以被重复使用，这样就可以用很小的数据容量来显示大尺寸的背景。接下来我们就来讲解如何将大地图用高效并快速的方式显示出来。

示例程序 Scroll_3_1.cpp 实现了大地图的显示。这个程序中重要的部分是 DrawMap 函数中的以下内容（代码清单 2-3-1）。

代码清单 **2-3-1**　显示地图的主要处理（**Scroll_3_1.cpp** 片段）

```
    fMap_x = VIEW_WIDTH / 2.0f - fCamera_x;
    fMap_y = VIEW_HEIGHT / 2.0f - fCamera_y;
    nBaseChip_x = ( int ) - fMap_x / CHIPSIZE;
    nBaseChip_y = ( int ) - fMap_y / CHIPSIZE;
    fBasePos_x = fMap_x + nBaseChip_x * CHIPSIZE;
    fBasePos_y = fMap_y + nBaseChip_y * CHIPSIZE;
    nChipNum_x = VIEW_WIDTH / CHIPSIZE + 1 + 1;
    nChipNum_y = VIEW_HEIGHT / CHIPSIZE + 1 + 1;
        fChipPos_y = fBasePos_y;
        for ( i = 0; i < nChipNum_y; i++ ) {
        fChipPos_x = fBasePos_x;
        for ( j = 0; j < nChipNum_x; j++ ) {
            DrawMapChip( fChipPos_x, fChipPos_y,
                         nMapData[nBaseChip_y + i
            fChipPos_x += CHIPSIZE;
        }
        fChipPos_y += CHIPSIZE;
    }
```

本次引用的代码段有点长，下面将依次说明。首先，变量 fMap_x 与 fMap_y 是假设地图为一张大图片时地图左上角的坐标。虽然最终画面上会绘制很多个地图块，但这些都是以地图左上角的坐标为基准的。这个程序中使用的地图块在水平方向有 MAPSIZE_X 个（实际是 21 个），在垂直方向有 MAPSIZE_Y 个（实际是 12 个）。简单地说就是，以 fMap_x 与 fMap_y 所示的左上角坐标为起点，在垂直方向循环 MAPSIZE_Y 次，水平方向循环 MAPSIZE_X 次，共绘制出（MAPSIZE_Y × MAPSIZE_X）个地图块就可以了（参考图 2-3-2）。

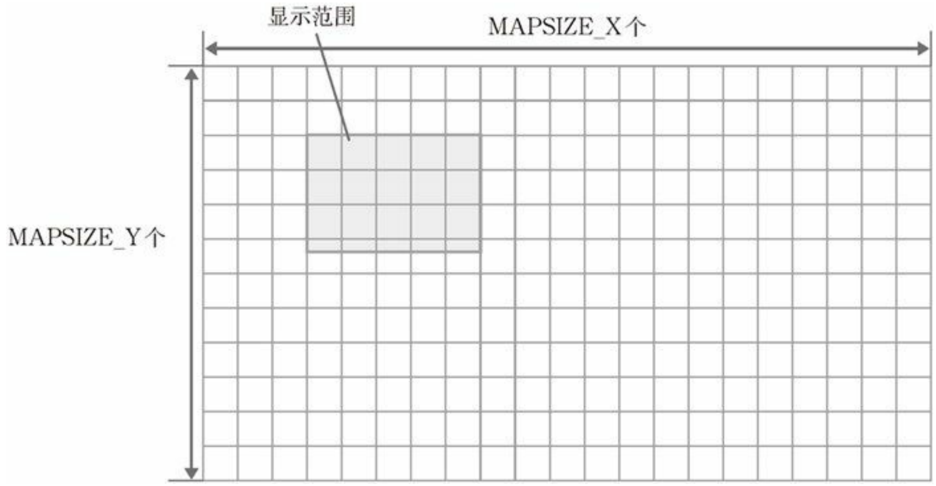


图 **2-3-2**　从左上角坐标开始循环配置地图块

事实上，虽然通过这种方式将地图块依次填充可以绘出一整张正确的地图，但是在正式的游戏中却很少这样做。因为在大部分游戏中，地图都是非常大的，画面所能显示的只是地图中很小的一部分，如果用地图块提前将整张地图都绘制出来，大部分地图都没有在画面上显示出来，也就失去了意义（参考图 2-3-2）。可能有人会想，反正迟早都要将地图绘制出来，提前绘制好有什么问题呢？这是由于无论是 2D 还是 3D 的绘图系统，大量无用的绘图命令都会浪费绘图系统的资源，这里所说的资源是指 CPU 的处理时间、内存容量等。在绘图系统中，只要发出一条绘图指令，即便实际上什么都没有绘制，也是相当花费时间的。另外，由于大多数绘图系统中都会使用显示列表或顶点缓冲的形式将需要显示的信息暂存起来，如果其中存储了太多实际用不到的图形，就会严重浪费内存。因此在实际绘制地图时，应该遵循一条原则，即只绘制需要显示的地图块。

在示例程序 Scroll_3_1.cpp 中，将需要绘制的地图块编号保存在了变量 nBaseChip_x 与 nBaseChip_y 中。具体来说，nBaseChip_x 中保存的是从地图左端起第一个需要绘制的地图块编号，nBaseChip_y 中保存的是从地图上端起第一个需要绘制的地图块编号（参考图 2-3-3）。

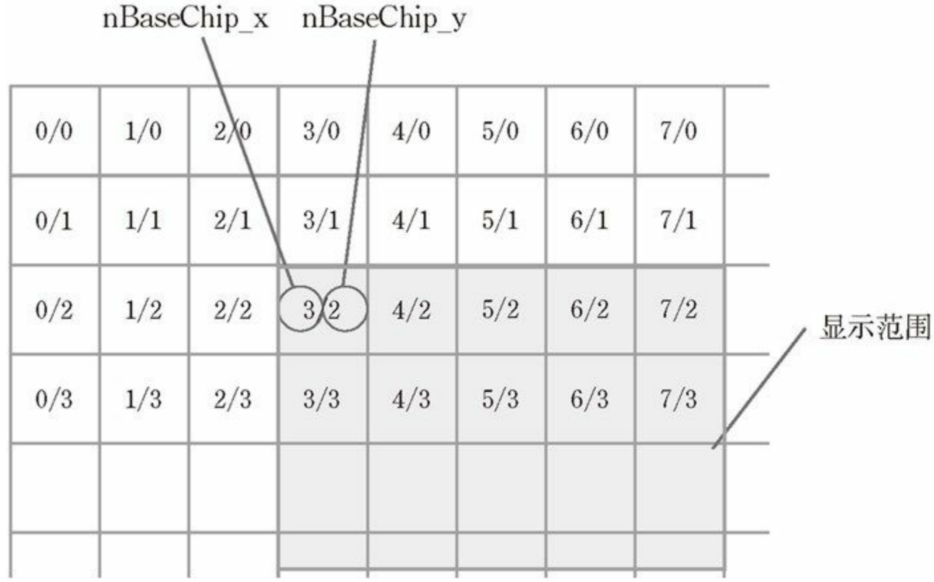


图 **2-3-3**　将显示范围中左上角的地图块编号存入变量

在程序中，nBaseChip_x 与 nBaseChip_y 实际上是通过以下方式计算得到的。

```
    nBaseChip_x = ( int ) - fMap_x / CHIPSIZE;
    nBaseChip_y = ( int ) - fMap_y / CHIPSIZE;
```

首先将地图左上角的坐标取负号再取整，然后除以地图块的大小（实际为 64）。通过对地图左上角坐标取负号，可以得到以地图左上角坐标为原点（0,0）时显示画面的左上角坐标。而对其取整，是由于表示地图块大小的 CHIPSIZE 是在程序最上方定义的整数型常数。而由于是在整数间进行除法运算，因此计算结果的小数点之后的部分都会被丢弃，其结果正好可以用来表示地图的总宽度除以地图块尺寸 CHIPSIZE 后，显示画面中左上角显示的是第几个地图块。由此我们就得到了需要绘制的地图块编号（参考图 2-3-3）。

顺便一提，计算 nBaseChip_x 与 nBaseChip_y 的语句

```
    nBaseChip_x = ( int ) - fMap_x / CHIPSIZE;
    nBaseChip_y = ( int ) - fMap_y / CHIPSIZE;
```

中，如果 CHIPSIZE 固定为 64，那么可以使用

```
    nBaseChip_x = ( int ) - fMap_x >> 6;
    nBaseChip_y = ( int ) - fMap_y >> 6;
```

获得同样的结果。这是一种可以替代除法运算的方法，称为“移位”。在以前 CPU 指令中还没有除法运算，或者除法运算特别花时间的时期多用这种手法。但是现在除法运算指令已经非常高速，而且在上式中除法运算也没有被嵌套在大量循环中，所以即便替换为移位也没有太大意义。现代的 CPU 中大都内置了名为“桶形移位器”的逻辑电路，可以在一个时钟周期内进行任意长度的移位，因此如果是在数量非常巨大的循环中，将除法运算替换为移位运算会获得更快的速度。

- 计算地图块的绘制位置

  上面已经计算出了需要绘制的最左上角的地图块编号，接下来就来计算一下画面上绘制该地图块的实际坐标。

```
    fBasePos_x = fMap_x + nBaseChip_x * CHIP
    fBasePos_y = fMap_y + nBaseChip_y * CHIP
```

这里仍然以大地图的左上角坐标为基准。由于目前画面所要显示的地图块，是地图中自上数第 nBaseChip_y 块，自左数第 nBaseChip_x 块，因此将其分别乘以每个地图块的尺寸，再加上大地图的左上角坐标（fMap_x, fMap_y），就可以得到实际需要绘制的地图块的坐标（参考图 2-3-4）。

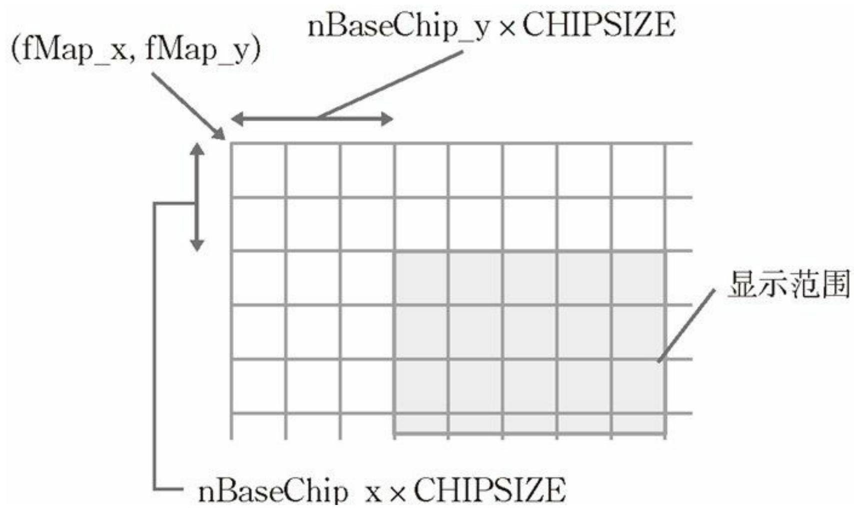


图 **2-3-4**　以大地图的左上角坐标为基准计算显示坐标

其实这里也有一个小窍门，如果 CHIPSIZE 为 64，那么坐标计算就可以写成

```
fBasePos_x = -( ( int )-fMap_x & 0x3f);
fBasePos_y = -( ( int )-fMap_y & 0x3f);
```

即通过逻辑运算（这里使用了 AND 运算符）替

代乘法运算来计算坐标位置。由于现在的 CPU 中乘法运算的速度与逻辑运算的速度的差别比起以前来已经非常小了，因此使用逻辑运算的必要性也变小了。不过即便乘法运算能在 CPU 的一个时钟周期内完成，但乘法运算和逻辑运算分别在 CPU 的不同单元中处理时，由于按顺序执行的乘法运算和逻辑运算可以并行处理，因此还不能说逻辑运算已经完全可以被替代。

POINT　绘制地图使用的乘法、除法运算可以被替换为更快的逻辑运算，不过平时使用乘法和除法就足够了。

下面让我们回到程序的处理中，来计算一下画面上需要绘制的地图块数量。

```
    nChipNum_x = VIEW_WIDTH / CHIPSIZE + 1 +
    nChipNum_y = VIEW_HEIGHT / CHIPSIZE + 1
```

一般情况下，在水平方向用画面宽度除以地图块尺寸，在垂直方向用画面高度除以地图块尺寸，应该就能得到要绘制的地图块数量。但是实际上这还不够，上面的程序中就将所得数值 +1+1，即额外多绘制了 2 个。

第一个 +1，是为了应对画面大小无法整除地图块的情况。如果画面大小不是地图块尺寸的整数倍，画面上就会有一部分溢出，即存在一些地图块只需要绘制一部分（参考图 2-3-5）。为了应对这些只显示一半的地图块，必须对绘制数 +1。

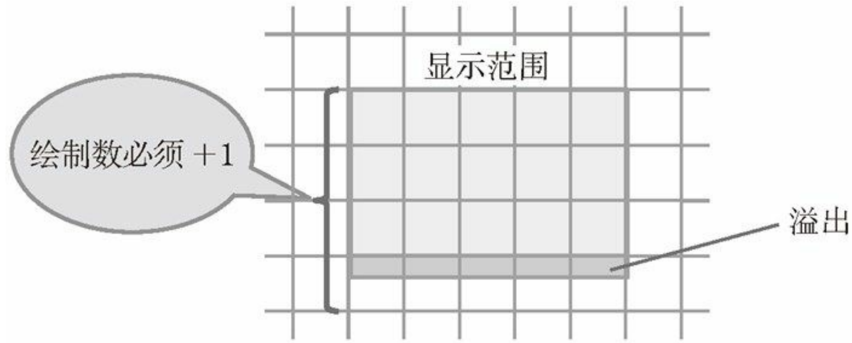


图 **2-3-5** 如果不绘制溢出部分的地图块，画面显示就会有问题

实际上在示例程序 Scroll_3_1.cpp 中，地图块的大小 CHIPSIZE 为 64，画面的宽度 VIEW_WIDTH 为 640，正好可以整除 CHIPSIZE，而画面的高 480 则无法整除 CHIPSIZE。因此在水平方向上，

```
    nChipNum_x = VIEW_WIDTH / CHIPSIZE + 1 +
```

就可以省略一个 +1，改为

```
    nChipNum_x = VIEW_WIDTH / CHIPSIZE + 1;
```

从显示结果来看是没有什么区别的。而在垂直方向上，

```
    nChipNum_y = VIEW_HEIGHT / CHIPSIZE + 1
```

如果省略一个 +1，变为

```
    nChipNum_y = VIEW_HEIGHT / CHIPSIZE + 1;
```

就会因绘制数量不足而导致显示出现问题。

代码中的第二个 +1，是为了保证地图在卷动过程中也能够绘制出足够的地图块。当需要绘制的地图块的左上角坐标与画面左上角重合时，这个 +1 是不需要的。而当需要绘制的地图块的左上角坐标与画面左端不一致时，画面左边和右边都会出现一部分地图块，此时地图块绘制数就必须再 +1 了（参考图 2-3-6）。

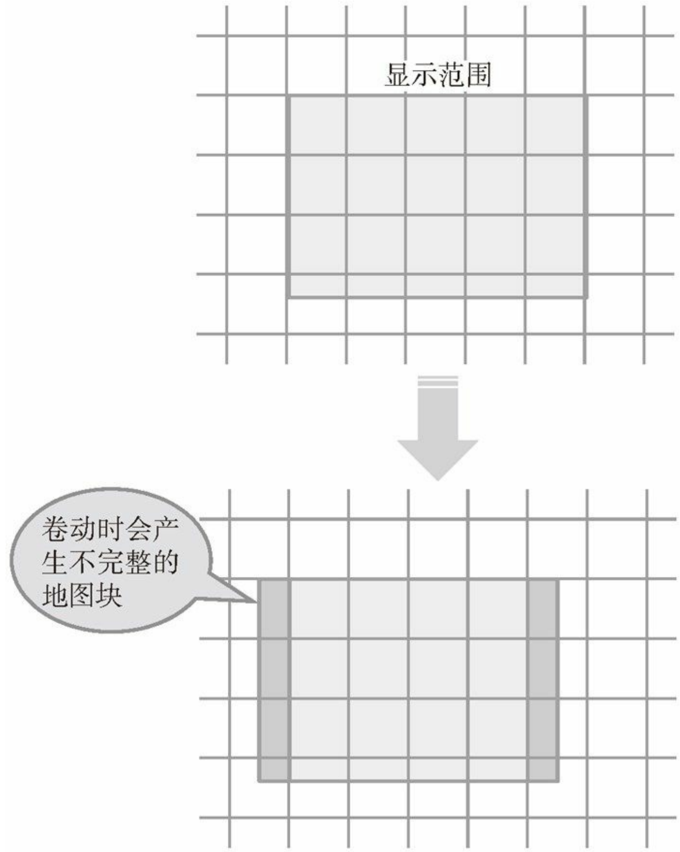


图 **2-3-6**　卷动时会出现不完整的地图块

在垂直方向上也出于同样的原因做了相同的处理，只不过由于地图块一开始就无法被画面高度整除，所以相比水平方向理解起来可能有点难。如果觉得在头脑中难以想象，可以在示例程序 Scroll_3_1.cpp 中删除一个 +1，然后观察程序实际的运行结果。

仔细观察一下下面两行计算绘制数量的部分，

```
    nChipNum_x = VIEW_WIDTH / CHIPSIZE + 1 +
    nChipNum_y = VIEW_HEIGHT / CHIPSIZE + 1
```

就会发现算式中包含的全部是类似VIEW_WIDTH、VIEW_HEIGHT、CHIPSIZE这样的常量，一个变量都没有。在这样的情况下，一般可以将整个语句使用define进行定义，而不必每次都重新计算，这样可以让程序更简短易懂。比如上面的语句可以写成

```
    #define CHIPNUM_X    ( VIEW_WIDTH  / CHIPSIZE +
    #define CHIPNUM_Y    ( VIEW_HEIGHT / CHIPSIZE +
```

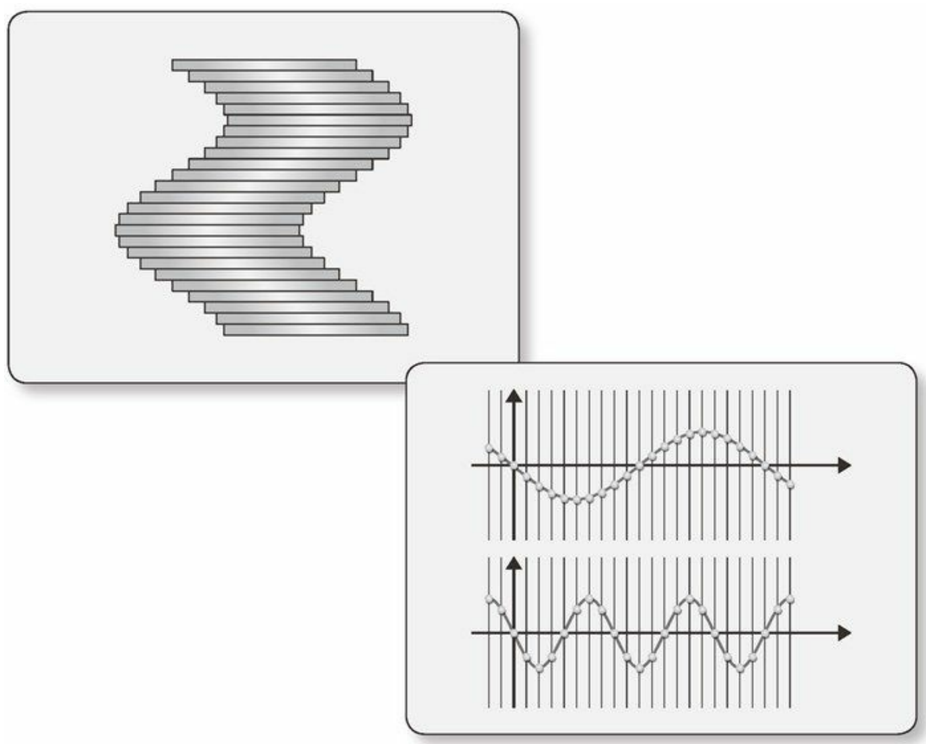
使用CHIPNUM_X与CHIPNUM_Y来替代nChipNum_x与nChipNum_y。至于在实际项目中是否将这样的语句转换为常量，则需要在综合考虑程序的易读性和工作量等的基础上进行判断。

在示例程序Scroll_3_1.cpp接下来的部分中，会使用目前为止得到的一些常量，通过一个2层循环将地图块实际绘制出来。而在这个过程中，当计算每个地图块的绘制坐标时，应该注意避免无用的乘法运算等，尽可能地优化代码效率。这些程序层面的技巧，可以在很多不同的情况下通用，具体实例请参考示例程序。

2.4　波纹式的摇摆卷动

Key Word　波纹扭曲、正弦波、波长、振幅、周期





卷动不光可以应用在背景的移动中，还可以用来实现一些动画效果。在本小节，我们就来一起学习卷动的应用。

前面已经学习了如何让整张图片沿同一方向卷动，本小节将讲解如何让一张图片的各部分通过卷动变形，并呈现波纹式的摇摆。

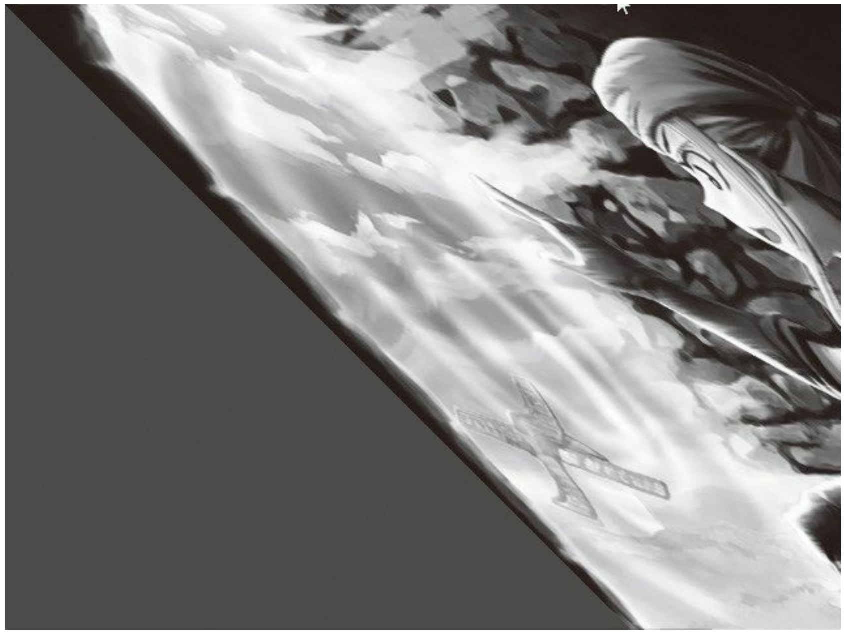


图 **2-4-1**　让图片呈 **45** 度扭曲倾斜的程序

首先看示例程序 Scroll_4_1.cpp，程序对一张图片进行了扭曲倾斜处理。程序中一个重要的部分，是在全局变量中定义的下面这行（代码清单 2-4-1）。

代码清单 **2-4-1**　图片扭曲倾斜程序的全局变量定义（**Scroll_4_1** 片段）

```
DRAWPOINT        v2Points[VIEW_HEIGHT];        //
```

在之后的程序中，会在 DirectX 渲染的部分读取这个变量，然后将图片沿垂直方向分割为 1 像素的细长线（这里称为图形线 ），并设置每一行图形线的绘制位置。即 v2Points[0].x = 0.0f、v2Points[0].y = 0.0f，v2Points[1].x = 0.0f、v2Points[1].y = 1.0f，v2Points[2].x = 0.0f、v2Points[2].y = 2.0f……当其中的 *x* 坐标都为 0 时，图片正常显示；而如果 *x* 坐标有值，图片就会根据坐标的不同产生扭曲（参考图 2-4-

2）。

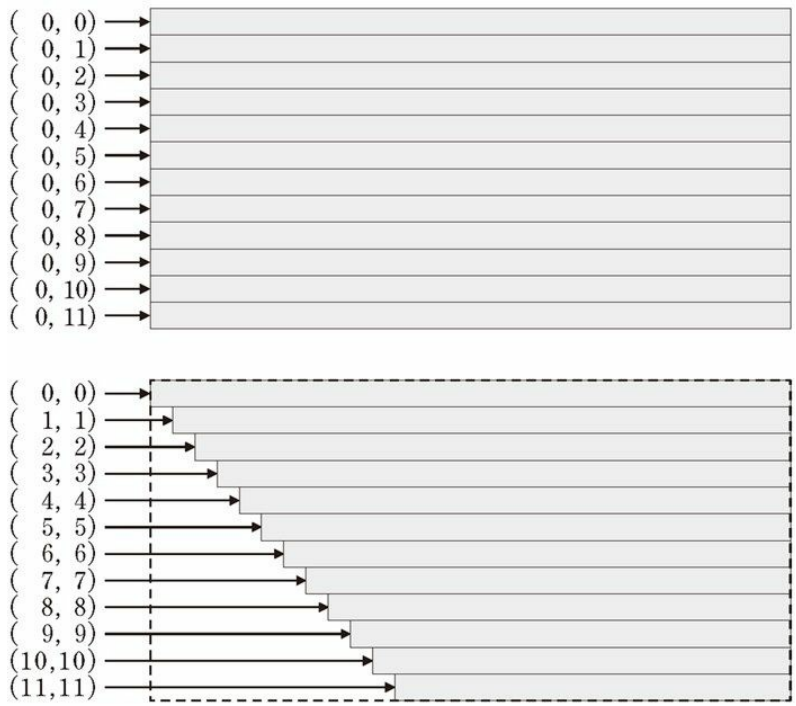


图 **2-4-2**　根据图形线绘制坐标的不同使图片产生扭曲

像这样根据每行图形线（也可以称作光栅 ）位置的不同进行卷动的方法，称为光栅扫描法 （其实只有在硬件层面上操作光栅才能称为真正的光栅扫描，本书中只是借用了这个概念）。

程序中另一个重要的部分，是 MoveBack 函数中决定每行图形线坐标的代码，如下所示。

代码清单 **2-4-2**　决定图形线坐标的部分（**Scroll_4_1.cpp** 片段）

```
    for ( i = 0; i < VIEW_HEIGHT; i++ ) {
        v2Points[i].y = ( float )i;    //图形线的
        v2Points[i].x = ( float )i;    //图形线的
    }
```

在上面的代码中，通过 for 语句执行 VIEW_HEIGHT 次（即图片的高度）循环，从而决定图片中每行图形线的位置。其中将第 i 像素的图形线绘制在从上开始第 i 行的代码是

```
        v2Points[i].y = ( float )i;       //图形
```

无论图片是否变形，这行代码都不需要更改，因为所有的图片都需要通过这样的处理绘制出来。

而程序中使图片变形的关键代码为

```
        v2Points[i].x = ( float )i;       //图形
```

即将第 i 行图形线的 *x* 坐标设置为 i，开始绘制图片时，图形线越靠下则 *x* 坐标越靠右，最终就绘制出了一张向斜 45 度方向扭曲的图片（参考图 2-4-2）。如果对此理解有困难的话，可以将

```
        v2Points[i].y = ( float )i;       //图形
        v2Points[i].x = ( float )i;       //图形
```

这部分更改为

```
        v2Points[i].y = ( float )i;       //图形
        v2Points[i].x = 0.0f;             //图形
```

这样图片就不会有任何变形了。而如果将代码更改为

```
        v2Points[i].y = ( float )i;       //图形
        v2Points[i].x = ( float )i * 0.5f; //图形
```

也可以观察图片实际会如何变形。建议大家实际验证一下，应该会更加容易理解。

- 使用正弦函数绘制正弦波

  在实现波纹式的动态卷动之前，让我们首先来实现让一张图片呈静态的波纹式扭曲。



图 **2-4-3**　将图片扭曲为波纹形状的程序

示例程序 Scroll_4_1a.cpp 实现了将图片扭曲为波纹形状的效果。这个程序中的重点是 MoveBack 函数中的以下部分。

代码清单 **2-4-3**　将图片扭曲为波纹形状的部分（**Scroll_4_1a.cpp** 片段）

```
        v2Points[i].y = ( float )i;
        v2Points[i].x = 30.0f * sinf( 2.0f *
```

与之前将图片呈 45 度扭曲倾斜的程序相比，这里绘制各图形线的 *y* 坐标部分没有改变，即在绘制图形线的 *y* 坐标时还是按照正常位置绘制。而针对图形线的 *x* 坐标，则使用了正弦函数（sin 函数）来决定其位置。如图 2-4-4 所示，正弦函数所绘制出的图形是平滑的波浪，所以可以实现类似波纹的扭曲效果。

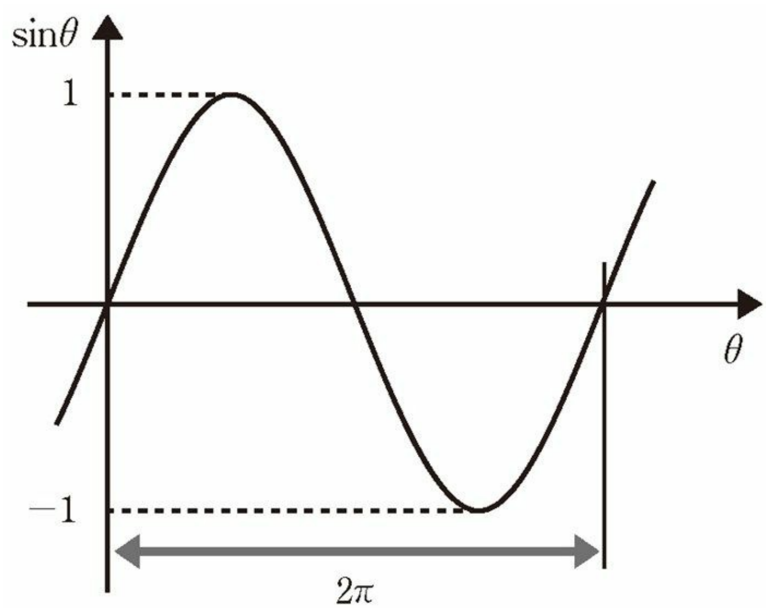


图 **2-4-4**　正弦函数所绘制的图形

像这样使用正弦函数生成的波形称为正弦波 。简单的正弦波可以用下面的公式表示。

$$y = A \cdot \sin\left(\frac{2\pi}{\lambda}x\right)$$

可以看到公式中的正弦波包含振幅 *A* 与波长 *λ* 两个特征。振幅 *A* 表示波振动的幅度大小，振幅越大说明波振动的幅度越大。具体来说，正弦函数的取值范围是 [-1, 1]，将其乘以常数 *A* 就可以得到振幅为 *A* 的波，所以振幅的取值范围是 [-A, A]（参考图 2-4-5）。

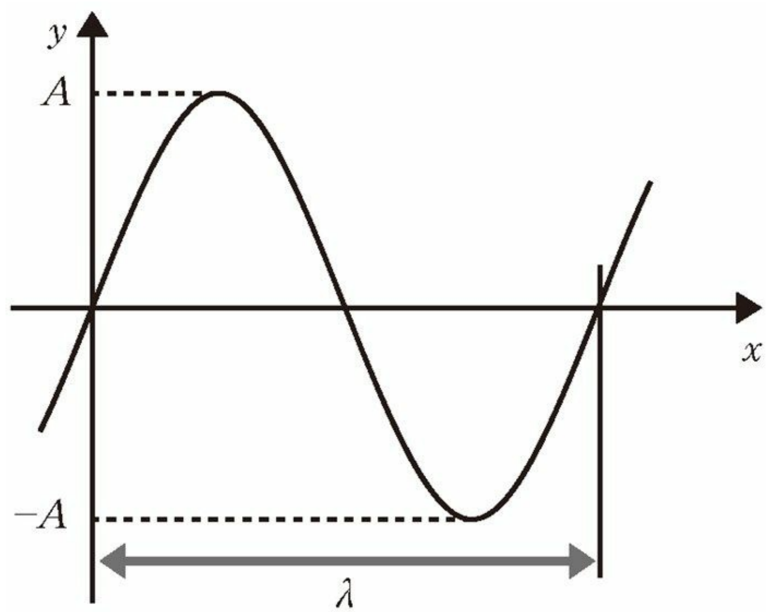


图 **2-4-5**　正弦波的振幅 ***A*** 与波长 ***λ***

波长 *λ* 表示波在 1 个周期内波动的距离，其值越大波形越宽（参考图 2-4-6）。

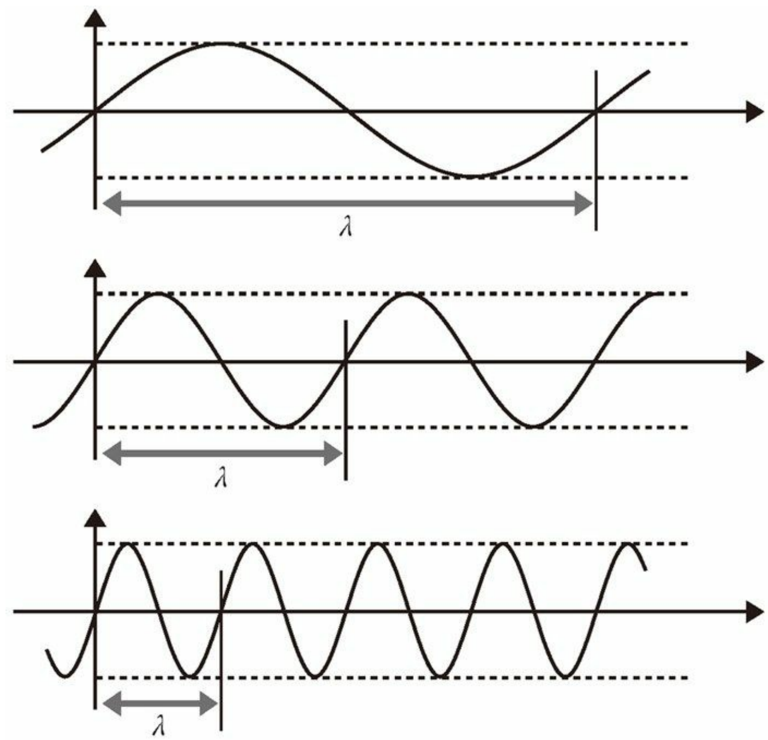


图 **2-4-6**　波在 **1** 个周期内的速度随着波长 ***λ*** 的大小而发生变化

如果 x 增加 λ，正弦函数就要增加 2π，而如果将角以弧度表示的话，2π 正好是 1 周期。让我们重新来看一下 Scroll_4_1a.cpp 中绘制图形线的 x 坐标的部分。

```
035  |        v2Points[i].x = 30.0f * sinf( 2.0f *
```

将其与正弦波的公式

$$y = A \cdot \sin\left(\frac{2\pi}{\lambda}x\right)$$

类比一下，可以得到

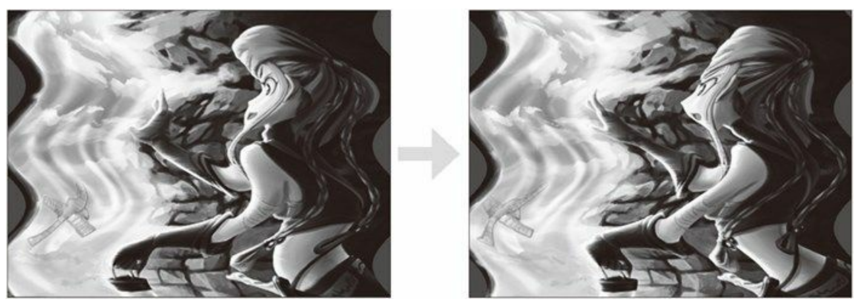
$A = 30$
$\lambda = 200$

即这个程序中使用了一个振幅为 30 像素、波长为 200 像素的正弦波来绘制图形线的 x 坐标，从而得到了波纹状的扭曲效果。

正弦波是游戏开发中经常用到的非常重要的一点，最好牢记其使用方法。

- 随时间动态摇摆

接下来，让我们尝试使图片随时间像波浪那样摇摆。



示例程序 Scroll_4_1b.cpp 实现了使图片像波浪那样摇摆的效果。程序中的重点是 MoveBack 函数中的以下部分。

代码清单 **2-4-4**　让图片像波浪那样摇摆的代码（**Scroll_4_1b.cpp** 片段）

```
        v2Points[i].y = ( float )i;
        v2Points[i].x = 30.0f * sinf( 2.0f *
```

对图形线的 y 坐标的处理仍然不变，而决定图形线的 x 坐标的正弦函数中则被加入了表示所经过

的时间的 fTime 变量。据此正弦函数就可以随时间产生变化，图片就会随时间进行摇摆。加入了时间要素的正弦波公式如下所示。

$$y = A\sin\left\{2\pi\left(\frac{t}{T} - \frac{x}{\lambda}\right)\right\}$$

变量 *t* 表示经过的时间。在这个正弦波公式中，除了之前出现过的振幅 *A*、波长 *λ* 外，还出现了周期 *T* 这一特征。所谓周期 *T*，正如其字面意思一样，表示多长时间是一个周期，其值越大，一次波动所经过的时间越长。上式中时间每推进 *T*，正弦函数中就增加 2π。让我们再重新看一下 Scroll_4_1b.cpp 中绘制图形线的 *x* 坐标的部分。

```
036  |        v2Points[i].x = 30.0f * sinf( 2.0f *
```

与包含时间的正弦波公式

$$y = A\sin\left\{2\pi\left(\frac{t}{T} - \frac{x}{\lambda}\right)\right\}$$

对比来看，可以得到

$A = 30$
$T = 60$
$\lambda = 200$

程序中 fTime 的时间单位为帧，因此图形线的 *x* 坐标将以振幅为 30 像素、周期为 60 帧（=1 秒）、波长为 200 像素的正弦波的形式摇摆。

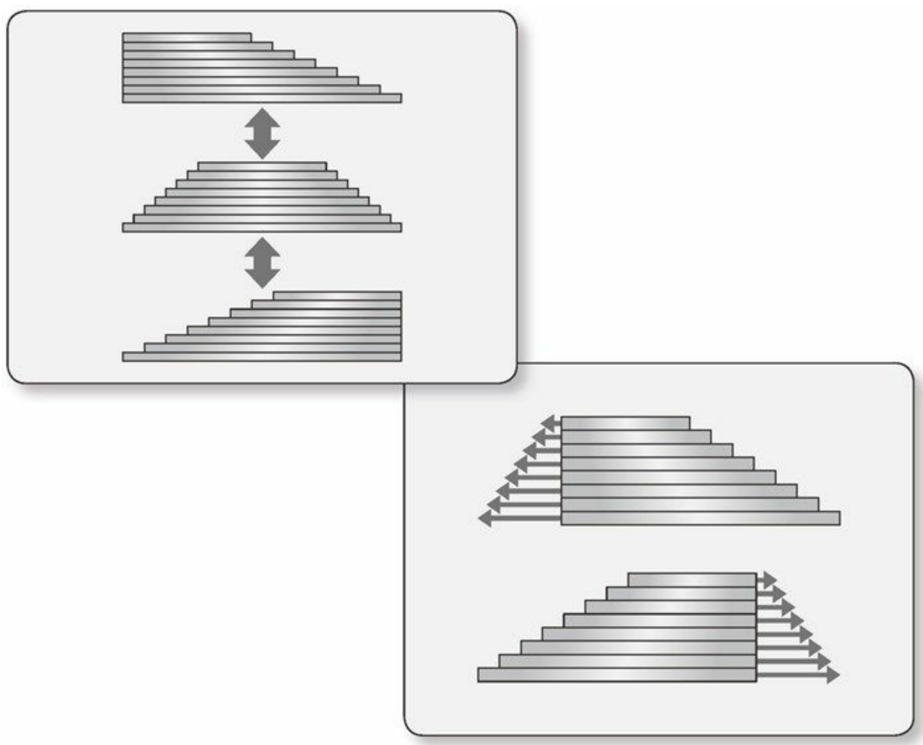
自然界中存在很多种波，与正弦波相似的波也有不少，因此通过正弦波就能大致表现出很多真实世界中的波（但是像水面那样的波严格来说并不

是正弦波）。如果读者对振幅 A 、波长 λ 、周期 T 这些常数的意义不太理解，建议尝试着修改一下程序中的这些常数，来实际观察波形的变化，从而更好地掌握。

2.5　制作有纵深感的卷动

Key Word　透视、比例计算、梯形

RANK/hard



通过将已经学习过的光栅扫描与多重卷动相结合，即便在 **2D** 绘图系统中，也可以做出有纵深感的画面效果。本小节就让我们一起来学习这些能提升虚拟世界的舞台效果的方法吧。

本小节中将不再使用 3D 多边形，而仅在 2D 绘图系统的范围内，制作出有纵深感（透视）的卷动效果。这种技术经常被用于展现格斗游戏的地面等。



图 **2-5-1**　用 **2D** 绘图表现带有纵深感的卷动程序

我们已经学习过多重卷动的实现，而为了制作出带有纵深感的卷动效果，就需要让近处地面的移动变快，远处地面的移动变慢。在使用前文介绍的图形线的方式来制作卷动效果的情况下，比如在表现地面时，可以使用一张梯形的背景，让下面的图形线快速移动，上面的图形线慢速移动，这样就可以做出有纵深感的地面了。示例程序 Scroll_5_1.cpp 就使用梯形背景实现了有纵深感的卷动。程序中的重点是 MoveBack 函数中的以下部分。

代码清单 **2-5-1**　实现有纵深感的卷动的程序的主要部分（**Scroll_5_1.cpp** 片段）

```
    fLineWidth = PIC_WIDTH_UP;
    fLineBase = ( PIC_WIDTH_DOWN - PIC_WIDTH_UP )
    for ( i = 0; i < VIEW_HEIGHT; i++ ) {
        v2Points[i].y = ( float )i;
        v2Points[i].x = fBack_x * ( fLineWidth -
        fLineWidth += ( float )( PIC_WIDTH_DOWN -
        fLineBase -= ( float )( PIC_WIDTH_DOWN -
    }
```

其中变量 fBack_x 是卷动的基准位置，即图形最下端的图形线，也是卷动速度最快部分的卷动位置。其他图形线的 *x* 坐标，都是以 fBack_x 为基准位置，通过一定的比例关系计算得到的。具体执行计算的是代码清单 2-5-1 中的

```
        v2Points[i].x = fBack_x * ( fLineWidth -
```

这一行。而具体是怎样计算的呢？让我们来分析一下。

为了让等式更容易理解，我们首先做以下假设（参考图 2-5-2）。

- 表示基准卷动位置的变量 fBack_x 为 x_B
- 正在处理的图形线中，表示有效图形（有效指梯形图片中需要被绘制的部分）开始的 *x* 坐标的变量 fLineBase 为 x_L
- 正在处理的图形线中，有效图形的宽度 fLineWidth 为 w_L
- 画面的宽度 VIEW_WIDTH 为 w_V
- 图形下端的宽度 PIC_WIDTH_DOWN 为 w_D
- 图形上端的宽度 PIC_WIDTH_UP 为 w_U

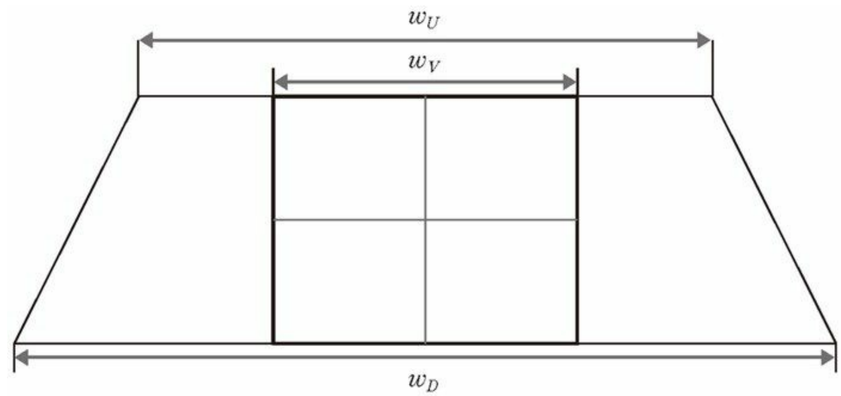


图 **2-5-2**　等式各要素与图形的对应关系

因此上面的程序语句就可以对应等式

$$x = \frac{w_L - w_V}{w_D - w_V} x_B - x_L$$

来分析一下这个等式。

首先考虑 $x_B = 0$ 即卷动的镜头在最左边时的情况，此时上面的等式为

$$x = - x_L$$

也就是说，当前图形线中有效图形的起始位置正好与画面左端一致，所有的图形线都会被绘制（参考图 2-5-3）。

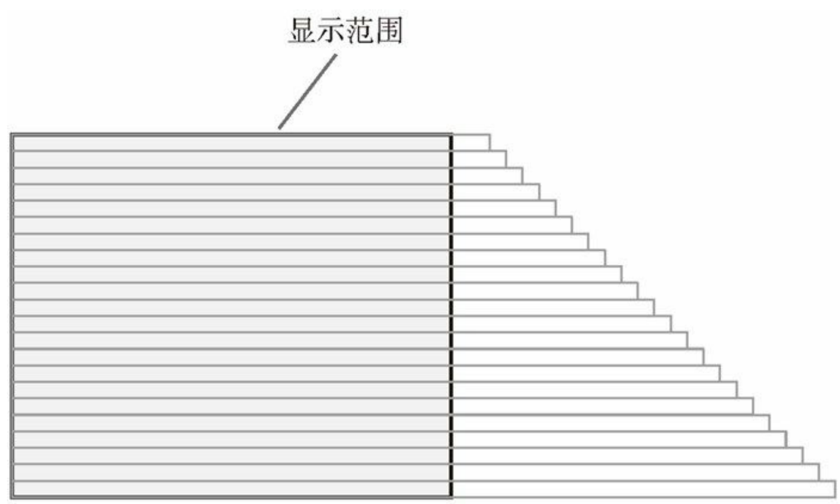


图 **2-5-3**　有效图形与画面左端一致

以这种情况为基准时，我们接下来要关注的就是，每一行图形线的最大卷动范围是多少。由于背景图是梯形，所以每行图形线可卷动的幅度都不同。于是就要使卷动幅度大的图形线快速卷动，幅度小的慢速卷动，从而形成最终的卷动效果。这与 2.1 节介绍的多重卷动中，宽度大的图形速度快，宽度小的图形速度慢的原理是一样的。将上述处理转换为公式，令图形的宽度为 w，画面的宽度为 w_V，那么可以卷动的最大幅度就为

$$w - w_V$$

而由于当前图形线中有效图形的宽度为 w_L，所以当前图形线可以卷动的最大幅度为

$$w_L - w_V$$

同理，在作为卷动基准的图形下端，因为有效图形的宽度为 w_D，所以图形下端可以卷动的最大幅度为

$$w_D - w_V$$

与多重卷动的原理相同，当前图形线与作为基准的下端图形线的卷动速度比为

$$\frac{w_L - w_V}{w_D - w_V}$$

因此，假设绘制当前图形线所需要的 x 坐标为 x，作为卷动基准的画面下端的图形线的 x 坐标为 x_B，则有

$$x = \frac{w_L - w_V}{w_D - w_V} x_B + C$$

其中 C 为常数，等于作为基准的 x_B 为零时 x 的值。

刚才已经得出一个结论，即 x_B =0 时镜头从最左边开始卷动，此时 x =-x_L 。所以常数 C =-x_L 。结果就得到了

$$x = \frac{w_L - w_V}{w_D - w_V} x_B - x_L$$

即程序中所书写的等式。

这样我们就得到了一行图形线的位置，因为最终需要绘制出整个梯形图形，所以需要根据每行图形线更改其宽度 fLineWidth（w_L ）以及当前图形线中有效图形起始位置的 x 坐标 fLineBase（x_L ）。fLineWidth 的计算方法是

```
065  |          fLineWidth += ( float )( PIC_WIDTH_DOWN -
```

即每行图形线都由上一行递增得到，递增的值是梯形的下底（PIC_WIDTH_DOWN）减去上底(PIC_WIDTH_UP) 后，再除以梯形中图形线的行数（VIEW_HEIGHT）而得到的结果。图形线从最开始的 PIC_WIDTH_UP 开始循环递增，经过 VIEW_HEIGHT 次循环后增加至 PIC_WIDTH_DOWN。

fLineBase 的计算方法为

```
066  |          fLineBase -= ( float )( PIC_WIDTH_DOWN -
```

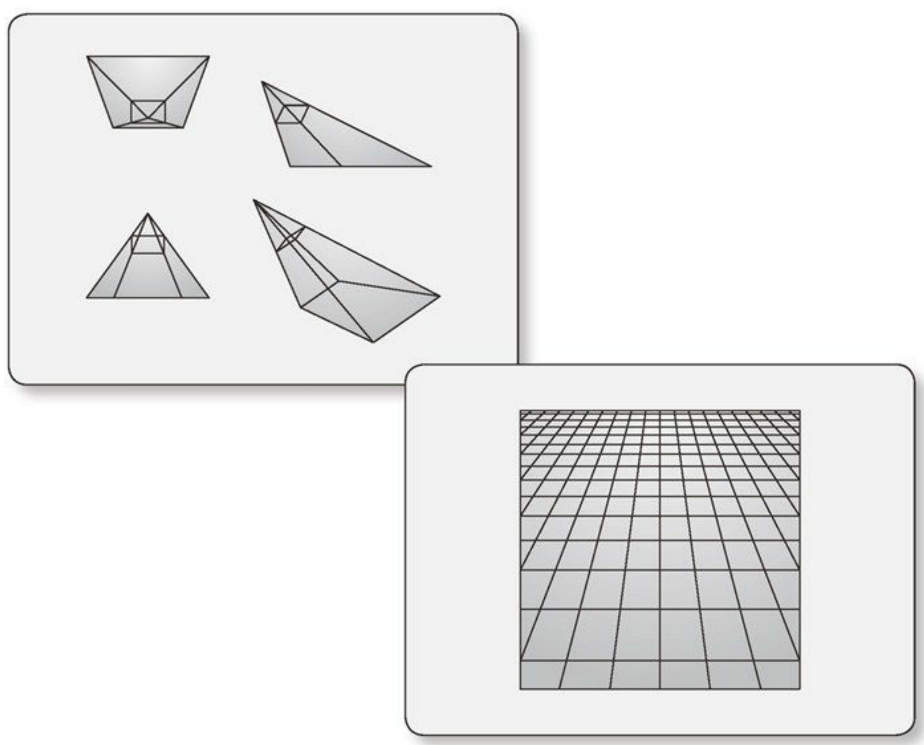
据此就可以沿梯形左侧的斜边向左方向移动。

这样我们就通过一张梯形图形以及图形线不同的移动速度实现了有立体效果的卷动，不仅没有使用任何 3D 计算，甚至连真正的 3D 绘图系统都没有用到。在 3D 绘图系统非常昂贵的时期，这无疑是一个提高游戏表现力的法宝。而在 3D 绘图已经比较普遍的今天，如果想让 2D 游戏多少也呈现出一些立体感的话，这也不失为一个简单的方法。

2.6　[进阶] 透视理论

Key Word　视景体（**view frustum**）、近似





上一小节中我们讲解了有纵深感的卷动，可以通过 **2D** 表现出类似 **3D** 的效果。本小节就让我们来更详细地了解一下这种效果背后的理论依据。

在 2.5 节中，我们使用一张梯形图片，并根据各行图形线的卷动速度的不同，实现了带有纵深感的卷动效果（也称为透视）。如果已经实际调试过有纵深感的卷动的示例程序 Scroll_5_1.cpp，就会发现其实现原

理并不难，但是却能表现出很好的立体卷动效果。本小节就让我们从理论角度论证一下 Scroll_5_1.cpp 的实现方式是否完备。

首先来思考一下真正的 3D 游戏中所使用的表现纵深感的方法。在真正的 3D 游戏中，会使用视景体 这一概念让画面产生纵深感。所谓视景体，就是将以视点（即玩家眼睛的坐标）为顶点、以显示器的屏幕为底面的四角椎体，沿显示器方向延长得到的图形（参考图 2-6-1）。

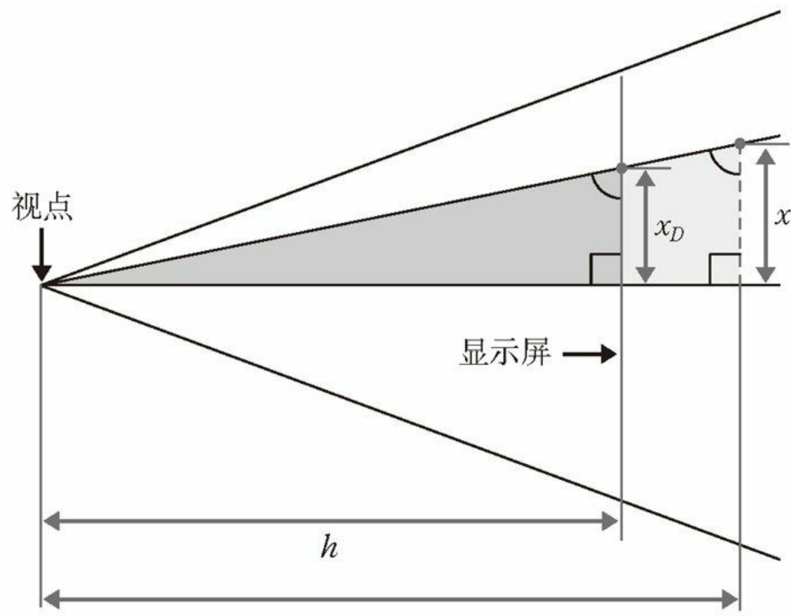


图 **2-6-1　3D** 中使用的视景体

想要得到虚拟空间中某个点在显示屏上的显示位置，只需要将该点与视点连接得到一条直线，然后计算出这条直线与屏幕的交点即可。这个交点的坐标就是该点在屏幕上的显示位置。通过这种方式，就可以简单并自然地通过计算实现远处物体小、近处物体大的显示效果。

具体来计算一下吧。假设显示屏与视点的距离为 h，视点的位置为原点 (0, 0, 0)。此时如果想要得知虚拟空间中的点 (x , y , z) 在显示屏上对应的显示位置，就

需要将空间中的点 (x, y, z)、显示屏以及视点构造一个如图 2-6-1 所示的三角形。例如，假设显示屏上的 x 坐标为 x_D，则有 $x_D : h = x : z$ 的比例关系成立。可得到

$$x_D \cdot z = h \cdot x$$
$$\therefore x_D = \frac{h \cdot x}{z}$$

同理，显示屏上的 y 坐标 y_D，也有

$$y_D = \frac{h \cdot y}{z}$$

即此时位于某坐标的点的 x 坐标与 y 坐标，在显示屏上变为了原来的 $\frac{h}{z}$ 倍。因为 z 坐标为分母，所以 z 坐标越大（即距离视点的深度越大），倍率越小，所显示的物体也就越小。请注意当 $z=h$，即 z 坐标正好与显示屏处于相同位置时，倍率正好为 1 倍，不会有放大或缩小。

3D 游戏就是基于这样的原理，来显示物体的放大倍数为 z 坐标的倒数，即 $\frac{1}{z}$ 倍的。假设透过我们的屏幕可以看到虚拟世界中一个水平的地面，那么屏幕上能显示地面多大的区域呢？从数学角度考虑的话，可以将这个问题转化为：用一个平面去切割代表视景体的四角椎体，求可以得到的图形。在上面的例子中，首先可以肯定的是，地面在屏幕上显示的区域，其边界肯定都为直线。这是因为视景体模型的四角椎体中，4 个侧面全部由平面构成，平面与平面相交的地方会形成直线。另外，如果切割四角椎体的平面与屏幕的上端（或下端）平行，那么这种情况就与带有纵深感的卷动相同，屏幕所对应的显示区域应该是一个梯形（参考图 2-6-2）。

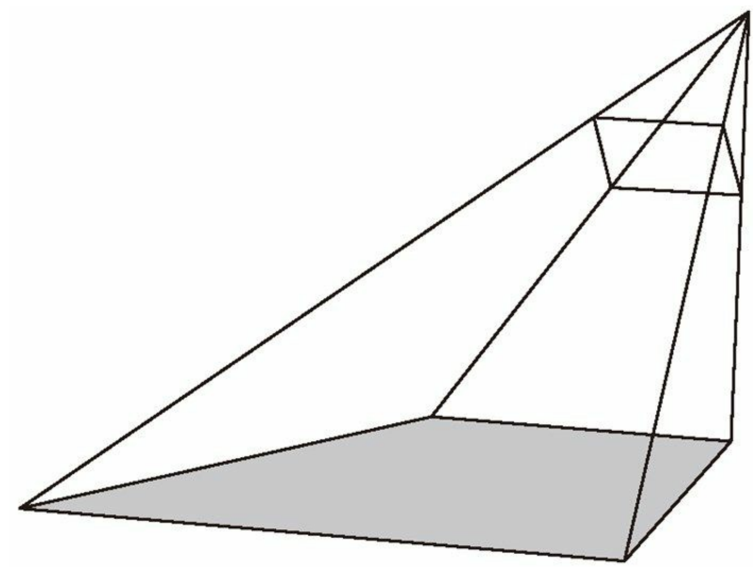


图 **2-6-2**　视景体的断面是梯形

需要注意的是，通过上例得到的梯形显示区域，是一个上底较长、下底较短的倒梯形。我们已经通过实例得知将视景体用一个平面切割会得到梯形，因此应该不难理解使用梯形的图片进行卷动，就可以让人产生纵深感。只是这种单纯使用梯形图片卷动与使用真正的 3D 所得到的结果并不完全一样。因为将屏幕上的图形线投影到实际的地面时，屏幕上方的图形线会变得疏松，下方的图形线会变得密集（参考图 2-6-3）。

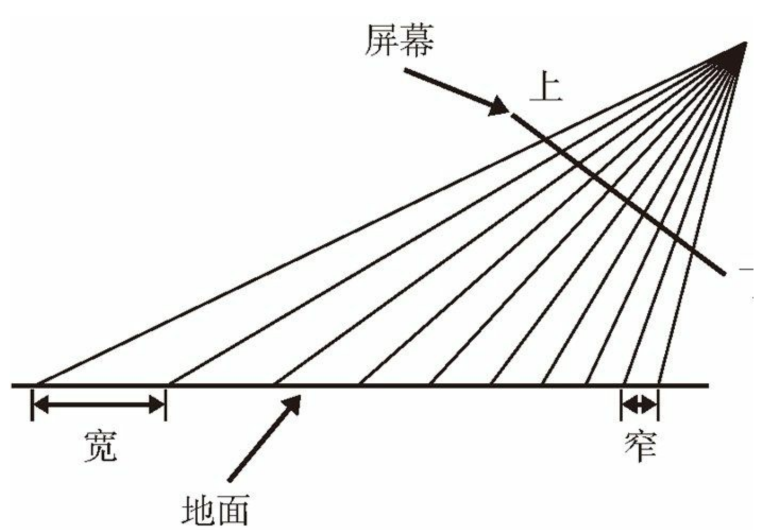


图 **2-6-3**　屏幕上方的图形线间距大，下方间距小

因此，如 2.5 节所示，将梯形图片简单地按垂直方向等间距切分为图形线并进行卷动，会让纵深感变得不明显。不过即便存在这些问题，对于大部分 2D 游戏来说，通过这种方式实现的立体感已经能派上很大用场了，一般也不会有人能注意到这样的细节。

事实上游戏世界中还有很多类似这样不易察觉的小问题。因为即便是 3D 游戏中所使用的用于表现纵深感的视景体，也只是一种现实世界的近似模型，多少也存在一些误差。而视景体模型的误差，主要是因为将观察者的视点看为了一个点。人类的眼睛，虽然称为视点，但其实并不是一个点，光线会首先穿过一定大小的瞳孔，然后透过晶状体聚焦，最后在半球形的视网膜上成像。因此如果将人眼的构造也考虑进来的话，当人眼位于显示屏的正对面一定距离时，所看到的图像是正确的，而用户一旦接近或远离屏幕，或者从某个方向斜视显示屏时，图像的正确性就无法保证，所以想要显示大多数情况下都正确的 3D 图像其实是非常难的。

通常在简单的 3D 显示中，只要不是那么吹毛求疵，视景体模型是足够用的。但在某些极端情况下，视景体模型的局限性就会暴露出来。比如现在的技术可以通过向两眼投影不同的图像，从而产生真实的立体效果，此时双眼会分别处理各自的视景体模型，然后通过调整视点的位置看到最终效果。向双眼投影不同图像时，必须综合考虑眼球转动的方向、瞳孔的大小、晶状体的焦距等因素，如果投影给用户的图像有偏差，用户的视觉系统就会为了消除视觉的不真实感进行高频率运动，严重时还会引发名为 3D 晕眩症的症状。

在不远的未来，可能会出现装载了更高技术水平的游戏，可以根据双眼的眼球位置、方向、瞳孔的大小、焦距的位置等，为左右两只眼睛准确地投影出最适合的图像。但就我个人而言，仅仅掌握基于视景体的立

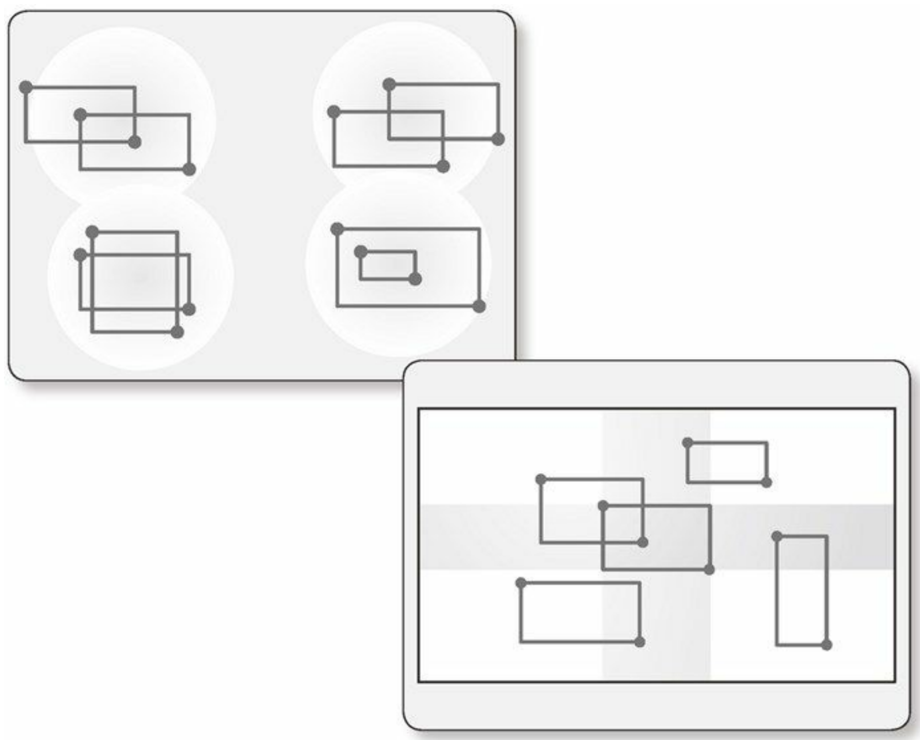
体效果就已经很吃力了，只能遥视那样先进的技术早日得以实现。

第 3 章　碰撞检测

3.1　长方形物体间的碰撞检测

Key Word　矩形、德摩根定律

RANK/easy



游戏开发中一个必不可少的要素当属碰撞检测。本小节将介绍长方形（矩形）物体间的碰撞检测的实现。

本章开始介绍游戏开发中必不可少的碰撞检测的实现，而本小节将首先介绍 2D 游戏中最基本的长方形（矩形）物体之间的碰撞检测。

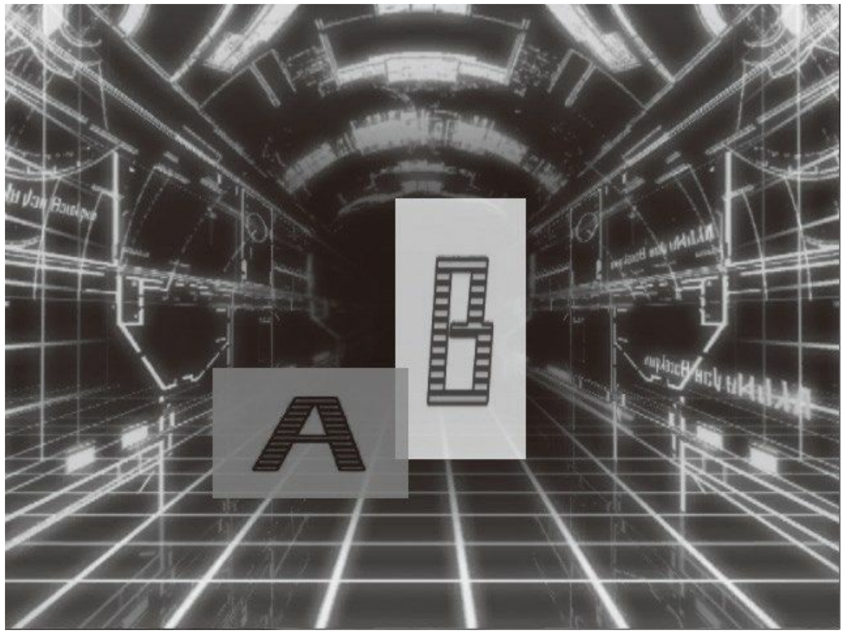


图 **3-1-1**　长方形物体间的碰撞检测

示例程序 CheckHit_1_1.cpp 实现了长方形物体间的碰撞检测。运行这个程序会显示“A”与“B”两个长方形，“A”长方形可以通过方向键移动，当其与“B”长方形重叠（即碰撞）时，颜色会变成红色。这个程序中执行碰撞检测的部分如代码清单 3-1-1 所示，会运行 CheckHit 函数。

代码清单 **3-1-1**　对于长方形物体间进行碰撞检测的 **CheckHit** 函数（**CheckHit_1_1.cpp** 片段）

```
int CheckHit( F_RECT *prcRect1, F_RECT *prcRect2
{
    int                 nResult = false;

    if ( ( prcRect1->fRight > prcRect2->fLeft ) &
         ( prcRect1->fLeft  < prcRect2->fRight )
    {
        if ( ( prcRect1->fBottom > prcRect2->fTop
             ( prcRect1->fTop    < prcRect2->fBot
        {
            nResult = true;
        }
    }

```

```
    return nResult;
}
```

CheckHit 函数以两个矩形（长方形）作为入口参数。代表矩形的 F_RECT 结构体的定义如下所示。

代码清单 **3-1-2　F_RECT** 结构体的定义（**CheckHit_1_1.cpp** 片段）

```
struct F_RECT {
    float          fLeft, fTop;          // 左、
    float          fRight, fBottom;      // 右、
};
```

结构体通过一个矩形左上角的坐标与右下角的坐标来描述一个矩形（参考图 3-1-2）。

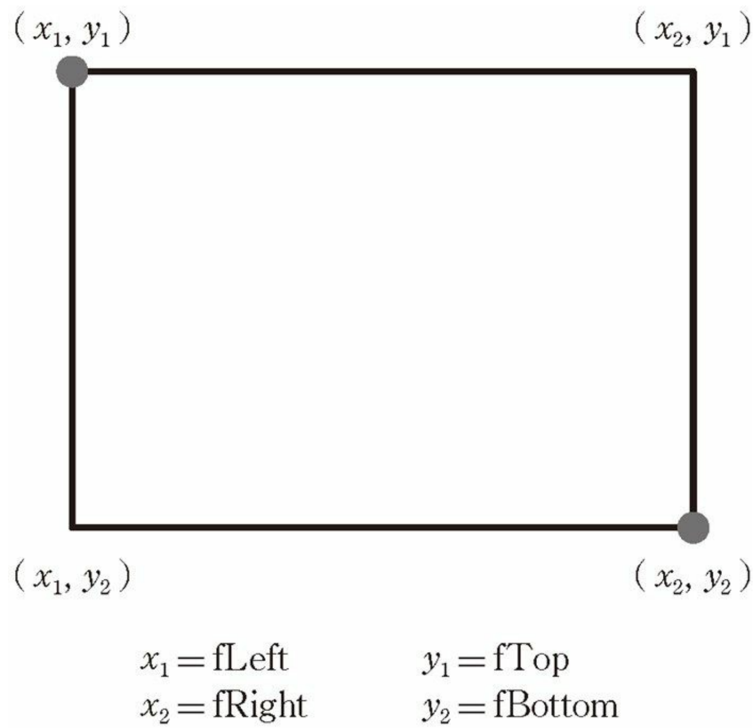


图 **3-1-2**　结构体 **F_RECT** 中使用了左上及右下的坐标

CheckHit 函数会对上述形式的两个矩形物体进行碰撞检测。检测过程中首先会通过下面两行代码，对水平方向上矩形重叠的可能性进行判定。

```
    if ( ( prcRect1->fRight > prcRect2->fLeft ) &
         ( prcRect1->fLeft  < prcRect2->fRight )
```

为了便于理解，我们先不要考虑什么情况下物体是可能重叠的，而是换个角度，想一想什么情况下两个物体是不可能重叠的。比如有矩形 1、矩形 2 两个矩形，以矩形 2 为基准（示例程序中是矩形 B），如果矩形 1 的右端比矩形 2 的左端还靠左，那么就可以认为两个矩形不可能重叠（参考图 3-1-3 左）。同理，如果矩形 1 的左端比矩形 2 的右端还靠右，也可以认

为两个矩形不可能重叠（参考图 3-1-3 右）。

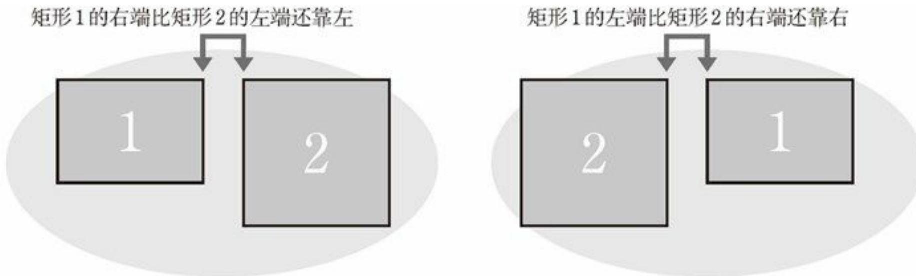


图 **3-1-3**　矩形没有重叠的情况

简而言之就是，当矩形 1 的右端比矩形 2 的左端靠左，或矩形 1 的左端比矩形 2 的右端靠右时，两个矩形是不可能重叠的。因此，在与上述结论相反的情况下，两个矩形就可能重叠，即矩形 1 的右端比矩形 2 的左端靠右 ，并且 矩形 1 的左端比矩形 2 的右端靠左 时，两个矩形是可能重叠的。上述理论，只要认真思考一下应该不难理解，在数学上称为德摩根定律。所谓德摩根定律，用程序员习惯的方式来书写的话，就是当有 bCond1 与 bCond2 两个条件时，

```
if ( !( bCond1 || bCond2 ) )
```

这一条件与

```
if ( !bCond1 && !bCond2 )
```

这一条件在任何情况下都是等价的。同时，

```
if ( !( bCond1 && bCond2 ) )
```

这一条件与

```
    if ( !bCond1 || !bCond2 )
```

这一条件在任何情况下也是等价的。用文字表述就是，对全体条件的否定，可以分解为对每个子条件的否定。只是这时逻辑运算 AND 要反转为 OR，OR 要反转为 AND。如果不熟记德摩根定律，虽然也可以通过逻辑推导得出同样的结论，但推导过程难免要占用时间，所以还是背下来比较有效率。

我们已经考虑了水平方向上重叠的情况，接下来对垂直方向也套用同样的方法，就可以得到结论：当矩形 1 的下端比矩形 2 的上端靠下，并且矩形 1 的上端比矩形 2 的下端靠上时，垂直方向上矩形 1 与矩形 2 是可能重叠的。执行这个检测的代码如下所示。

```
        if ( ( prcRect1->fBottom > prcRect2->fTop
             ( prcRect1->fTop    < prcRect2->fBot
```

当然，上述检测运行的前提条件，是在水平方向的检测中已经判断得出两个矩形是可能重叠的（如果没有这个前提，上面的 if 语句也不会执行）。因此当垂直方向的条件也满足时，矩形 1 与矩形 2 就是重叠的。此时，

```
                nResult = true;
```

变量 nResult 被置为 true，并作为 CheckHit 函数的返回值返回。而如果水平方向和垂直方向的条件都不满足，矩形则是没有重叠的，变量 nResult 的定义部分

```
    int                 nResult = false;
```

中，变量 nResult 的初始值被设置为 false，并且直接作为返回值返回。

在上面的讨论中，并没有考虑两个矩形正好在边界碰撞（或者说接触）的情况。比如上文中我们将与“矩形 1 的右端比矩形 2 的左端靠左”相反的情况视为“矩形 1 的右端比矩形 2 的左端靠左”，而这两者都漏掉了一种情况，即“矩形 1 的右端与矩形 2 的左端正好在同一位置”，因此当满足这个边界条件时应当如何处理，就需要好好考虑一下。

而在上文的示例程序 CheckHit_1_1.cpp 中，CheckHit 函数并没有处理两端正好位于同一位置的情况。这是由于考虑到矩形是基于左上角及右下角的坐标绘制的，而在现代的 3D 硬件中，这样的矩形的右端及下端的最后 1 像素正好会被省略。在现代的 3D 硬件中，比如要绘制左上角为（10，10）、右下角为（20，20）的矩形时，最左端的第 10 像素会被绘制，而最右端的第 20 像素一般则不会被绘制出来。垂直方向也一样，上端的第 10 像素会被绘制，而最下端的第 20 像素一般则不会被绘制。至于为什么有

这样的设定，我也没有详细了解过，但按照这样的设定，并且考虑到让碰撞检测的可视化效果更加真实，当端与端位于同一位置时，我们人为设定其为没有碰撞。读者可以将 CheckHit_1_1.cpp 中矩形 A 的移动速度调慢（比如 1 帧 1 像素），并仔细观察就会发现，当两个矩形重叠时会判定为碰撞，如果只是边缘接触则会判定为没有碰撞。另外读者也可以修改 CheckHit 函数中的 if 语句，全部加入等号，即

```
    if ( ( prcRect1->fRight >= prcRect2->fLeft )
         ( prcRect1->fLeft  <= prcRect2->fRight )
    {
        if ( ( prcRect1->fBottom >= prcRect2->fTo
               ( prcRect1->fTop    <= prcRect2->fBo
```

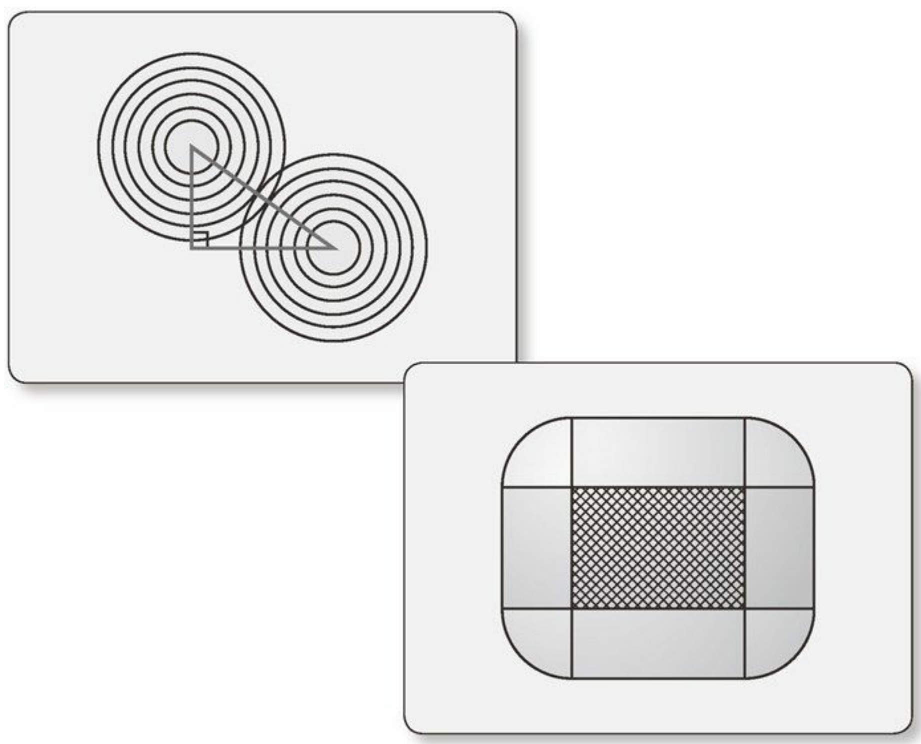
这样就可以让端与端位于同一位置时也判定为碰撞，那么当两个矩形仅仅边缘接触时，也满足碰撞检测的条件。示例程序 CheckHit_1_1.cpp 中是没有等号的，所以端与端在同一位置时就会判定为没有碰撞。

不过上面的处理方式还不能作为开发类似程序的统一标准，因为在众多绘图硬件及程序库中，也许存在特例会将图形的右端及下端也进行绘制。因此端与端在同一位置的碰撞检测，还是需要根据具体情况来决定如何处理。

3.2　圆形与圆形、圆形与长方形物体间的碰撞检测

Key Word　距离、勾股定理、平方比较





游戏中的碰撞检测，并不只限于矩形物体。根据实际需求，有时候也需要对圆形物体进行碰撞检测。本小节就让我们一起来学习圆形与圆形、圆形与长方形物体的碰撞检测。

本小节将介绍圆形与圆形、圆形与长方形物体间进行

碰撞检测的方法。

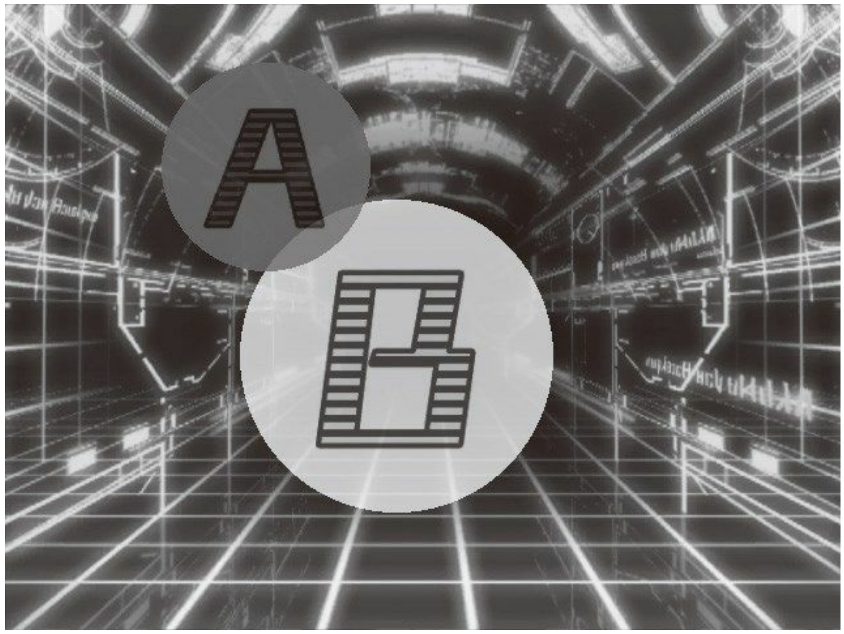


图 **3-2-1**　圆形物体间的碰撞检测

2D 游戏中经常会用到 3.1 节介绍的矩形物体间的碰撞检测。这主要是由于矩形物体间的碰撞检测通过一些简单的计算就可以实现，在一些低端机器上也能流畅地运行。而如今 PC 及游戏机都已经迈入高性能时代，即便碰撞检测中包含一些稍复杂的运算，大部分情况下也不会有什么问题。特别是圆形物体间的碰撞检测，由于计算方法相对简单，在 2D 游戏中也会经常用到。

那么接下来就让我们来看一下圆形物体间的碰撞检测。实际执行的程序为示例程序 CheckHit_2_1.cpp。程序中重要的部分是 CheckHit 函数的内容，如代码清单 3-2-1 所示。

代码清单 **3-2-1**　对圆形物体进行碰撞检测的 **CheckHit** 函数（**CheckHit_2_1.cpp** 片段）

```
int CheckHit( F_CIRCLE *pcrCircle1, F_CIRCLE *pcr
{
    int              nResult = false;
```

```
    float           dx, dy;    // 位置坐标之差
    float           ar;        // 两圆半径之和
    float           fDistSqr;

    dx = pcrCircle1->x - pcrCircle2->x;      //




    dy = pcrCircle1->y - pcrCircle2->y;      //




    fDistSqr = dx * dx + dy * dy;            //
    ar = pcrCircle1->r + pcrCircle2->r;
    if ( fDistSqr
```

其中被作为 CheckHit 函数的参数所传递的 F_CIRCLE 结构体如下所示。

代码清单 **3-2-2　F_CIRCLE** 结构体的定义（**CheckHit_2_1.cpp** 片段）

```
struct F_CIRCLE {
    float           x, y;                // 圆心
    float           r;                   // 半径
};
```

像这样，程序会通过两个圆的圆心及半径，来检测两个圆是否有碰撞。

至于具体要如何计算，可能数学好的朋友会想到：检测两个圆是否有交点，可以联立两圆的方程式，然后根据判别式求解。其实大可不必如此麻烦。判断两圆是否有碰撞，只要将两圆圆心之间的距离与两圆的半径之和相比较即可（参考图 3-2-2）。

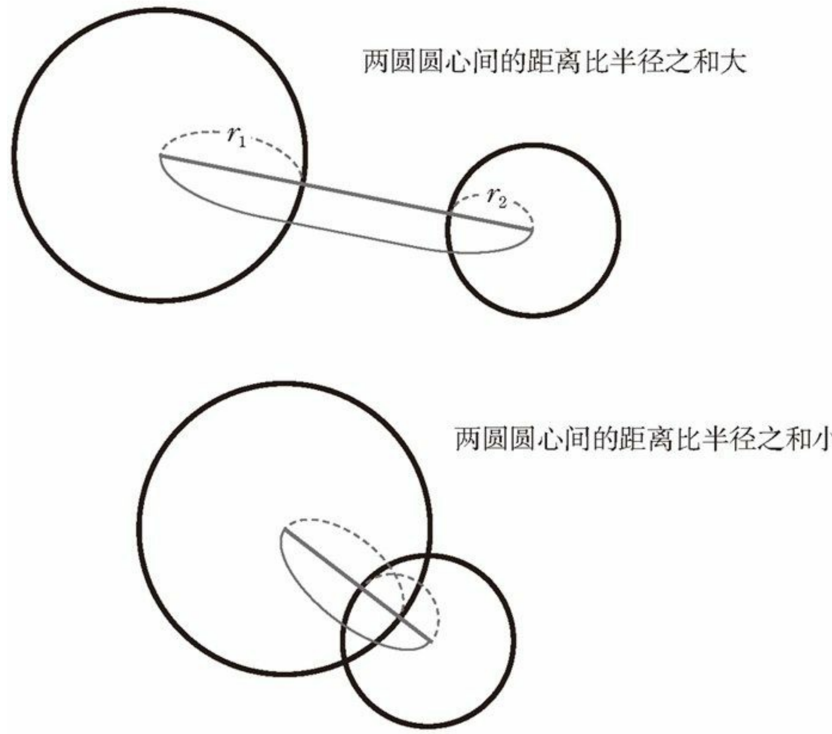


图 **3-2-2**　通过比较两圆圆心间的距离与两圆半径之和来进行碰撞检测

如此一来，就无需再考虑一个圆是否完全包含另一个圆或者两圆间是否有交点等情况，只需要一个条件就可以进行圆形物体间的碰撞检测。

那么就让我们来实际比较一下两圆圆心间的距离与两圆半径之和吧。假设圆 1 的圆心为 (x_1, y_1)，圆 1 的半径为 r_1，圆 2 的圆心为 (x_2, y_2)，圆 2 的半径为 r，两圆圆心间的距离为 l，根据勾 股定理有下列等式成立（参考图 3-2-3）。

$$l_2 = (x_1 - x_2)^2 + (y_1 - Y_2)^2$$

$$\therefore l = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

然后再将 l 与两圆的半径之和 $r_1 + r_2$ 比较，如果小于则可判定为碰撞。

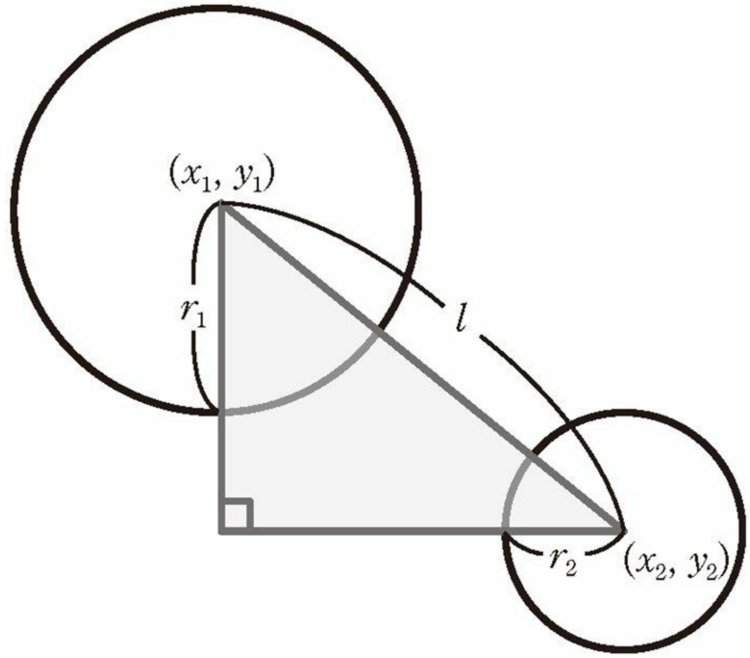


图 **3-2-3**　根据勾股定理计算两圆圆心距离

在示例程序 CheckHit_2_1.cpp 的 CheckHit 函数中，为了节省计算时间，并没有完全按照上面的等式书写语句，而是使用了略有不同的判定方法。可以看到在示例程序的 CheckHit 函数中，并没有计算平方根的 sqrtf 函数。这是因为该程序并没有直接使用圆心间的距离 l，而是通过比较距离的平方来进行的碰撞判定。这使用了数学中的当 $a_1 \geq 0$、$a_2 \geq 0$ 时，如果 $a_1 < a_2^2$，则必有$a_1 < a_2$ 这一原理（可参考小节末的 POINT 部分）。因为参与比较的对象为距离，都为非负数，所以距离之间的比较，与距离的平方之间的比较，所得到的结果是一样的。这个结论在游戏的碰撞检测中有时会用到，请读者最好能记下来。

> **POINT**　由于距离不可能为负数，所以距离之间的比较，与距离的平方之间的比较，其结果是一样的，直接比较平方还能节省计算时间。

将平方之间的比较写成语句，首先当

$$l < r_1 + r_2$$

时，两圆碰撞。由于 $l \geq 0$ 且 $r_1 + r_2 \geq 0$，因此分别对两边平方，得到

$$l^2 < (r_1 + r_2)^2$$

由勾股定理可知$l^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2$，将其代入上式，得到

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 < (r_1 + r_2)^2$$

即得到了 CheckHit 函数中的条件语句。

通过这种方式来回避平方根运算是非常有意义的，因为通过对平方进行比较，可以将平方根运算替换为乘法运算。而平方根运算的速度依赖于 CPU 的浮点数运算能力，与乘法运算所花费的时间有巨大差别。根据硬件结构的不同，两者的差距有时可达 10 倍以上，因此凡是能回避平方根运算的地方，都应该尽可能地在程序中使用上述方法。

- 圆形与长方形的碰撞检测

  接下来就让我们来学习圆形与长方形的碰撞检测。由于圆形与长方形的形状存在差异，这类碰撞检测会稍微复杂一些。最终实现的程序请参考示例程序 CheckHit_2_2.cpp。这个程序中重要的部分是 CheckHit 函数的内容，由于程序比较长，这里就不直接引用代码了。本小节仅从理论层面讲解一下实现的思路，具体代码请读者自行查阅示例程序（源代码下载地址请参考文前的“关于本书”）。

  在 CheckHit_2_2.cpp 的 CheckHit 函数中，主要可以分为以下两个阶段进行碰撞检测。

  1. 将需要检测的长方形，在上下左右4个方向均向外扩张，扩张的长度为圆半径r，如

果扩张后得到新的长方形内包含了圆心坐标，则认为两物体具备碰撞的可能（反之则无碰撞的可能）。

2. 在满足条件 1 的情况下，如果圆心坐标在原长方形以外、扩张后的长方形的左上、左下、右上、右下四个角处，且圆内没有包含长方形最近的顶点，则认为两物体没有碰撞。

步骤 1、2 的判定请参考图 3-2-4。之所以通过这种方式检测，是因为按照一般的思维方式，仅仅考虑长方形的各边是否与圆形相交，是不够全面的，还必须考虑到圆形完全包含长方形，或长方形完全包含圆形等情况。因此在 CheckHit_2_2.cpp 中，并没有去检查长方形的各边与圆周是否相交，而是针对长方形包含圆形这一特殊情况，先进行了条件 1 的检测，然后又针对圆形包含长方形的情况，进行了条件 2 的检测。

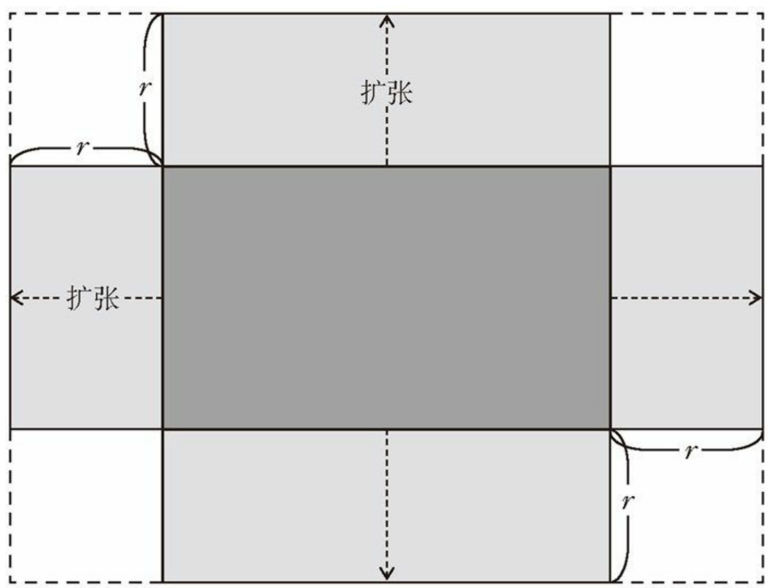


图 **3-2-4**　将长方形向外扩张圆半径 ***r*** 来进行碰撞检测

具体来说，首先条件 1 的检测是以长方形为中心来考虑的。之所以将各边向各方向扩张半径 r 形成新的长方形，是因为圆的半径为 r，当圆心在原长方形之外，且与长方形的距离为 r 时，圆与长方形也有可能发生碰撞。但是扩张后的长方形的四个角，即边长为 r 的正方形区域则需要特殊判断（参考图 3-2-4）。因为在这个区域中包含圆心的情况下，仍然存在圆的上下左右端都与原长方形不相交的可能性，这时就需要根据条件 2 进行判断。条件 2 的检测是以圆形为中心的，从这个角度出发来考虑圆形不包含长方形的情况应该更加容易理解。判断圆形不包含长方形，或者以圆形为中心检测时判断二者没有发生碰撞的最简单的条件是：长方形的 4 个顶点中，离圆心最近的顶点没有在圆内。即当满足条件 1 而不满足条件 2 时，就可以认为两物体没 有碰撞。之所以按这样的顺序进行两个阶段的判定，是为了尽可能地减少计算量。首先进行简单的判定，将明显不是碰撞的情况排除，并判断是否有必要进行有距离（确切来说是距离的平方）计算的条件 2 的判定，而只有在必要时，才会进行最终的计算处理。

POINT

当 $a_1 \geq 0$ 且 $a_2 \geq 0$ 时，如果 $a_1^2 < a_2^2$ 则 $a_1 <$ a_2 的证明

$$a_1^2 < a_2^2$$

将 a_2^2 移项到左边得到

$$(a_1^2 - a_2^2) < 0$$

将左边因数分解得到

$$(a_1 + a_2)(a_1 - a_2) < 0$$

由于 $a_1 \geq 0$ 且 $a_2 \geq 0$ 时，$a_1 + a_2 \geq 0$，这样一个正值乘以 $(a_1 - a_2)$ 为负值，所以 $(a_1 - a_2)$ 为负值，即

$$(a_1 - a_2) < 0$$

将 a_2 移项到右边得到

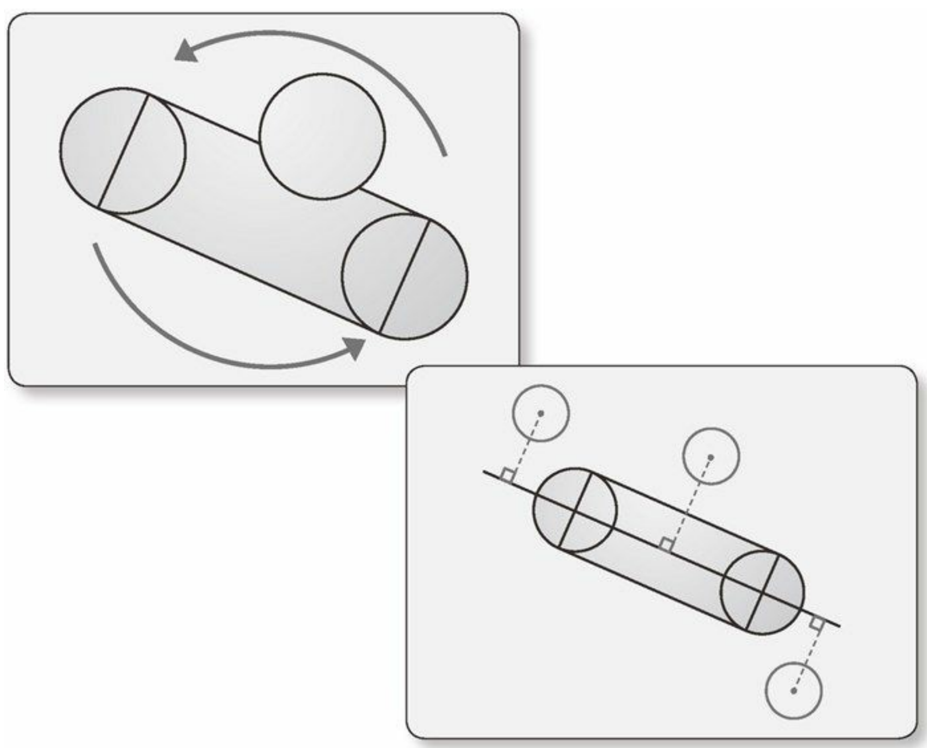
$$a_1 < a_2$$

即得到证明。另外，根据 $y = x^2$ 这个函数在 $x \geq 0$ 的范围内是单调递增函数，也可以证明当 $a_1 \geq 0$ 且 $a_2 \geq 0$ 时，如果 $a_1^2 < a_2^2$，则 $a_1 < a_2$。

3.3 细长形物体与圆形物体间的碰撞检测

Key Word 点与线段的距离、内积、微分

RANK/hard



接下来让我们一起学习圆形与细长形物体的碰撞检测。在游戏中，细长形物体可以被用于显示激光或剑等。

本小节将讲解圆形与细长形物体（如激光或剑）的碰撞检测。比如从某个斜方向发射激光等时，如果仍然

使用 2D 碰撞检测中的常规手段，将光线作为长方形物体处理的话，就会非常麻烦，所以需要对细长形物体做特殊处理。



图 **3-3-1**　细长形物体与圆形物体的碰撞检测

下面就来一起看一下细长形物体与圆形物体间的碰撞检测吧。请参考示例程序CheckHit_3_1.cpp。程序中重要的部分是 CheckHit 函数的内容，如代码清单 3-3-1 所示。

代码清单 **3-3-1**　执行细长形物体与圆形物体碰撞检测的 **CheckHit** 函数（**CheckHit_3_1.cpp** 片段）

```
int CheckHit( F_CIRCLE *pcrCircle, F_RECT_CIRCLE
{
    int         nResult = false;
    float       dx, dy;           // 位置坐标之差
    float       t;
    float       mx, my;           // 对应最短距离
    float       ar;               // 两物体的半径

    float       fDistSqr;

    dx = pcrCircle->x - prcRectCircle->x;
```

```
    dy = pcrCircle->y - prcRectCircle->y;
    t = ( prcRectCircle->vx * dx + prcRectCircle-
        ( prcRectCircle->vx * prcRectCircle->vx +
    if ( t 0.0f="" )="" t="0.0f;">t
的下限
    if ( t > 1.0f ) t = 1.0f;                // t
的上限
    mx = prcRectCircle->vx * t + prcRectCircle->x
    my = prcRectCircle->vy * t + prcRectCircle->y
    fDistSqr = ( mx - pcrCircle->x ) * ( mx - pcr
               ( my - pcrCircle->y ) * ( my - pcr
    ar = pcrCircle->r + prcRectCircle->r;
    if ( fDistSqr
```

函数接收两个参数，一个是代表圆的 F_CIRCLE 结构体指针，另一个是代表由一个长方形和两端的两个半圆形组成的细长形物体的 F_RECT_CIRCLE 结构体指针。当两者碰撞时，返回值为 true。F_CIRCLE 结构体通过圆心坐标与半径来表示一个圆，F_RECT_CIRCLE 结构体则通过一个点以及一个向量来表示一个线段，同时该结构体还包含到此线段的有效距离，最终就可以表示一个倾斜的长方形两端附加半圆形的细长图形（参考图 3-3-2）。

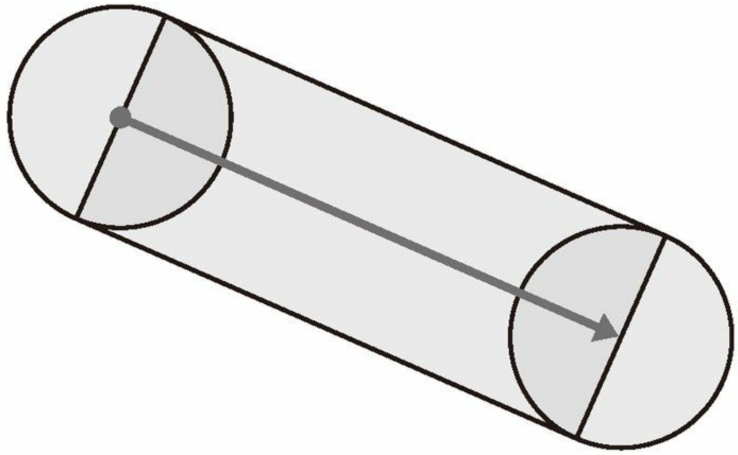


图 **3-3-2　F_RECT_CIRCLE** 结构体表示的图形

为了检测这两个图形是否碰撞，首先需要计算 F_CIRCLE 结构体的圆心坐标到 F_RECT_CIRCLE 结构体中的线段的最短距离；然后再计算“F_CIRCLE 结构体的圆半径 + 到 F_RECT_CIRCLE 结构体中的线段的有效距离”，如果最短距离小于后者，则可认为两者碰撞。用语言表达可能会比较难理解，这里我们重新用数学算式来表达，设圆心与线段的最短距离为 *l* min，圆的半径为 r_1，到线段的有效距离为 r_2（参考图 3-3-3），当满足

$$l_{\mathrm{lim}}lt;r_1+r_2$$

时，两物体碰撞。

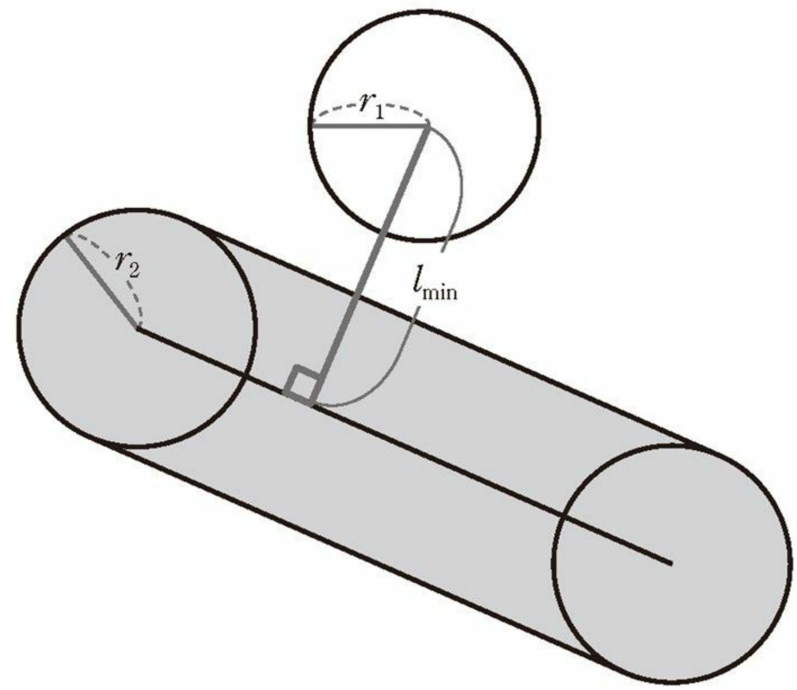


图 **3-3-3**　通过圆心坐标与到细长形物体的线段的最短距离进行碰撞检测

这里面的问题是，点与线段的最短距离 l min 要如何计算。如果将点与线段的距离更改为点与直线的距离，则可以利用下面这个有名的公式。

直线 $ax+by+c=0$ 与点 (x_0, y_0) 的最短距离为

$$\frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

但是目前我们想要计算的，并不是点到一条无限延长的直线的距离，而是点到一条有起点和终点的线段的距离，因此无法直接套用上面的公式。当然如果肯花时间，也可以使用上面的公式得到一个可行的算法，但是我们不打算这样做。这是因为公式中直线是使用 $ax+by+c=0$ 的形式来表示的，而在程序中，为了适应更多的场景，直线往往都使用向量的形式来表示，即用向量 $\boldsymbol{a}$ 、 $\boldsymbol{b}$ 将直线上的位置向量 $\boldsymbol{p}$ 表示为

$$\boldsymbol{p} = \boldsymbol{a}\,t + \boldsymbol{b}$$

这样就可以适用于更多的场景。如果想从事游戏开发，就应该学会用向量这种形式来表示直线。

POINT 游戏编程中经常使用向量来表示直线。

可能有人已经注意到了，传给 CheckHit 函数的参数 F_RECT_CIRCLE 结构体，正是使用向量形式表示的线段。其中位置向量 $\boldsymbol{b}$ 表示直线所通过的一点的位置，向量 $\boldsymbol{a}$ 表示直线延伸的方向（参考图 3-3-4)。

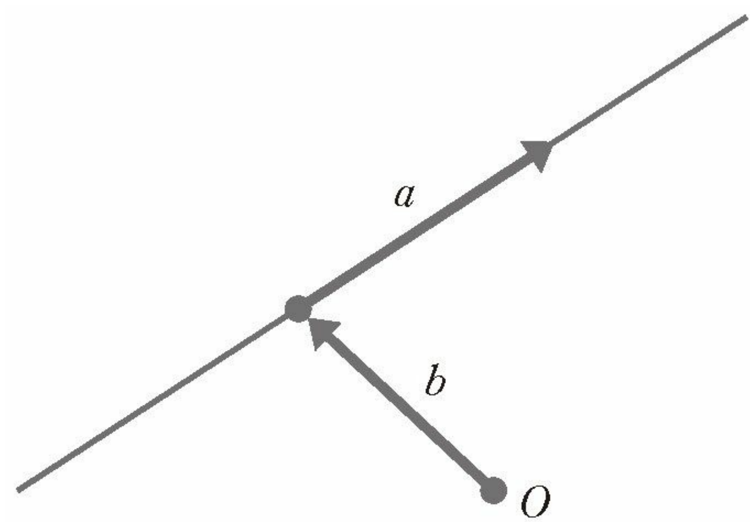


图 **3-3-4 F_RECT_CIRCLE** 结构体中线段的表示

但是上面的式子只能表示一条无限延伸的直线，因此我们认为 CheckHit 函数中还隐含了条件 $0 \leq t \leq 1$。即结构体最终表示的是以位置向量 $\boldsymbol{b}$ 与位置向量 $\boldsymbol{b} + \boldsymbol{a}$ 为两端的线段。此时该线段与点 (x_0, y_0) 的最短距离应该如何计算呢？可以从以下两点考虑。

1. 将线段上的点 $\boldsymbol{p} = \boldsymbol{a}t + \boldsymbol{b}$ $(0 \leq t \leq 1)$ 与点 (x_0, y_0) 的距离表示为 t 的函数，然后对 t 进行微分，求得距离最小时的 t。

2. 过点 (x_0, y_0) 向包含线段的直线 $\boldsymbol{p} = \boldsymbol{a}t + \boldsymbol{b}$ 做一条垂线，使用向量的内积求得距离的最小值 t

。

下面首先介绍方法 1。此时令线段上的点 $\boldsymbol{p}$ 为 (p_x, p_y)，向量 $\boldsymbol{a}$ 为 (a_x, a_y)，向量 $\boldsymbol{b}$ 为 (b_x, b_y)， $\boldsymbol{p} = \boldsymbol{a}t + \boldsymbol{b}$ $(0 \le t \le 1)$ 可以分解表示为

$$\begin{cases} p_x = a_x t + b_x \\ p_y = a_y t + b_y \end{cases} \quad (0 \le t \le 1)$$

设此线段上的点与点 (x_0, y_0) 的距离为 l，根据勾股定理有

$$\begin{aligned} l^2 &= (p_x - x_0)^2 + (p_y - y_0)^2 \\ &= (a_x t + b_x - x_0)^2 + (a_y t + b_y - y_0)^2 \\ &= a_x^2 t^2 + 2a_x(b_x - x_0)t + x_0^2 + a_y^2 t^2 + 2a_y(b_y - y_0)t + y_0^2 \\ &= (a_x^2 + a_y^2)t^2 + 2\{a_x(b_x - x_0) + a_y(b_y - y_0)\}t + x_0^2 + y_0^2 \end{aligned}$$

根据我们之前得到的结论，可知当 $l_1 \ge 0$ 且 $l_2 \ge 0$ 时，如果 $l_1^2 \ge l_2^2$ 则必有 $l_1 \ge l_2$。而这里的距离 l 必定是大于 0 的，所以如果 $l_1^2 \ge l_2^2$ 则必有 $l_1 \ge l_2$ 这个结论成立。换言之，求距离平方最小值对应的 t，等同于求距离最小值所对应的 t。因此上式

$$l^2 = (a_x^2 + a_y^2)t^2 + 2\{a_x(b_x - x_0) + a_y(b_y - y_0)\}t + x_0^2 + y_0^2$$

中，令 l^2 最小的 t，就是点到线段的最短距离所对应的 t。为了求 l^2 的最小值，将等式用 t 进行微分，得到

$$\frac{\mathrm{d}(l^2)}{\mathrm{d}t} = 2(a_x^2 + a_y^2)t + 2\{a_x(b_x - x_0) + a_y(b_y - y_0)\}$$

要求使 $\frac{\mathrm{d}(l^2)}{\mathrm{d}t}$ 为零的 t 的值，需要让 l^2 取极值（极大值或极小值），在上式中应该为极小值。因为对 l^2 进行 t 的 2 阶微分（即对上式再进行一次微分），可得

$$\frac{\mathrm{d}^2(l^2)}{\mathrm{d}t^2} = 2(a_x^2 + a_y^2)/ > 0$$ 0">　（不过向量 ***a*** 并不是零向量）

等式必然为正，因此函数的 2 阶微分为正时所取的极值为极小值。

综上，通过 $\frac{\mathrm{d}(l^2)}{\mathrm{d}t}$ 为零的 *t* 的值，我们就可以计算出线段与点的最短距离。下面让我们通过如下步骤求得 *t*
。

$$\frac{\mathrm{d}(l^2)}{\mathrm{d}t} = 2(a_x^2 + a_y^2)t + 2\{a_x(b_x - x_0) + a_y(b_y - y_0)\} = 0$$
$$2(a_x^2 + a_y^2)t = -2\{a_x(b_x - x_0) + a_y(b_y - y_0)\}$$
$$\therefore t = \frac{a_x(x_0 - b_x) + (a_y(y_0 - b_y)}{(a_x^2 + a_y^2)}$$

这就是 CheckHit 函数中

```
    dx = pcrCircle->x - prcRectCircle->x;
    dy = pcrCircle->y - prcRectCircle->y;
    t = ( prcRectCircle->vx * dx + prcRectCircle-
        ( prcRectCircle->vx * prcRectCircle->vx +
```

这部分等式的由来。请注意程序中的变量与数学等式的对应关系为

$$a_x = \mathrm{prcRectCircle} - / > \mathrm{vx}$$
$$a_y = \mathrm{prcRectCircle} - > \mathrm{vy}$$
$$(x_0 - b_x) = \mathrm{dx}$$
$$(y_0 - b_y) = \mathrm{dy}$$

\text{vx}\\&a_y=\text{prcRectCircle}->\text{vy}\\&(x_0-b_x)=\text{dx}\\&(y_0-

b_y)=\text{dy}\end{aligned}">

但是这样计算出的 t 并不一定满足线段的条件 $0 \leq t \leq 1$。因此在 CheckHit 函数中，如果 t 没有在 0 到 1 的区间内，会通过下面的方式将 t 收敛在 0～1 的范围内。

```
    if ( t 0.0f="" )="" t="0.0f;">t
的下限
    if ( t > 1.0f ) t = 1.0f;             // t
的上限
```

即当 t 比 0 小时令 t 等于 0，当 t 比 1 大时令 t 等于 1。这也说明了求线段与点的最短距离 t 时，l^2 是 t 的二次函数，其极小值只有一个。

- 使用求得的 ***t*** 计算线段与点的最短距离

  到目前为止，求得的还不是线段与点的最短距离，而是可以令线段与点有最短距离的 t。CheckHit 函数会继续使用 t 计算线段与点的最短距离（确切说是最短距离的平方）。为此首先需要通过以下方法求得有最短距离的线段上的坐标。

```
    mx = prcRectCircle->vx * t + prcRectCirc
    my = prcRectCircle->vy * t + prcRectCirc
```

  这是将 t 代入线段的等式

$$\begin{cases} p_x = a_x t + b_x \\ p_y = a_y t + b_y \end{cases} \quad (0 \leq t \leq 1)$$

得到的。接下来使用勾股定理，通过如下方法求得最短距离的平方。

```
    fDistSqr = ( mx - pcrCircle->x ) * ( mx
               ( my - pcrCircle->y ) * ( my -
```

将所得最短距离的平方（即变量 fDistSqr 的值），与“F_CIRCLE 结构体的圆半径 + 到 F_RECT_CIRCLE 结构体中的线段的有效距离”的平方相比较，如果前者小于后者，则认为细长形物体与圆碰撞。这在程序中体现为以下的 if 语句，如果碰撞则将表示结果的变量 nResult 置为 true。

```
    ar = pcrCircle->r + prcRectCircle->r;
    if ( fDistSqr < ar * ar ) {            /
        nResult = true;
    }
```

而由于变量 nResult 在声明时已经设默认值为 false，如果上面的 if 语句不满足，则变量 nResult 仍返回 false。最终当碰撞时 nResult 为 true，未碰撞时 nResult 为 false，并通过以下语句作为函数的返回值返回。

```
    return nResult;
```

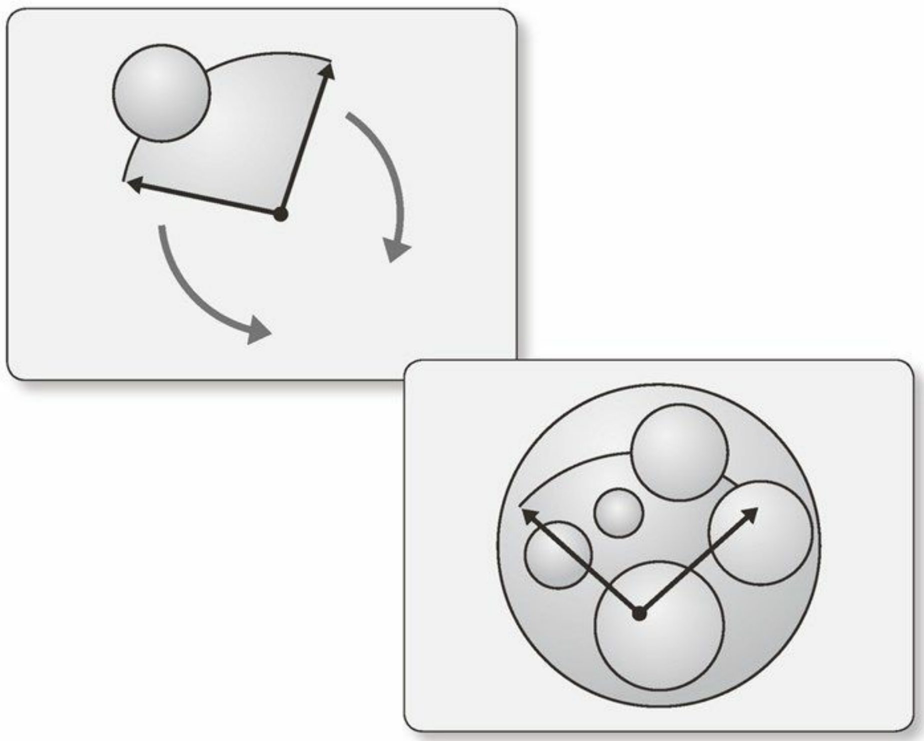
本节我们介绍了两端为圆形的细长形物体的碰撞检测，之所以选择这样一个有点奇怪的图形，而不使用倾斜的长方形，是出于一些考量的。首先，倾斜的长方形与圆形的碰撞检测会有非常多的条件分支，导致程序很长。对于惯用指令流水线（instruction pipeline）和推测执行（speculation execution）的现代 CPU 来说，过多的分支条件不利于速度的优化。其次，使用斜长方形作为碰撞检测的区域时，最终呈现的效果可能会有一些不合理的地方（即便从数学的角度来看是正确的）。特别是当另一个图形擦过长方形的四个角时，这种情况会被认为发生了碰撞，而肉眼看上去却是没有碰撞的，这样就在无形中给玩家带来了困扰。因此为了让细长物体的两端更好地符合现实世界的碰撞现象，特意将四角处理为了圆形。

类似本小节这样的复杂图形的碰撞检测判定，基本上很难在大脑中计算整个过程，需要在纸上一步步地演算。而真正的游戏开发，更是不仅要能在纸上完成演算过程，还要能将其代码化，有意成为游戏开发者的读者朋友，请以此为目标努力吧。

3.4　扇形物体的碰撞检测

Key Word　条件划分、向量的运算、向量的内分点、圆的方程式

RANK/hard



游戏都是以角色为中心的，角色与一定区域的碰撞检测时有发生，而检测区域则以扇形居多。与扇形的碰撞检测中，条件划分是极为重要的。

本小节将讲解扇形物体与圆形物体的碰撞检测，这在挥舞剑等细长物体时的碰撞检测等中经常会用到。



图 **3-4-1**　扇形物体与圆形物体的碰撞检测

示例程序 CheckHit_4_1.cpp 实现了扇形物体的碰撞检测。程序中重要的部分是 CheckHit 函数的内容，如代码清单 3-4-1 所示。

代码清单 **3-4-1**　执行扇形物体与圆形物体碰撞检测的 **CheckHit** 函数（**CheckHit_4_1.cpp** 片段）

```
int CheckHit( F_CIRCLE *pcrCircle, F_FAN *pfaFan
{
    int             nResult = false;

    float           dx, dy;
    float           fAlpha, fBeta;
    float           fDelta;
    float           ar;
    float           fDistSqr;
    float           a, b, c;
    float           d;
    float           t;

    dx = pcrCircle->x - pfaFan->x; dy = pcrCircle
    fDistSqr = dx * dx + dy * dy;
    if ( fDistSqr < pcrCircle->r * pcrCircle->r )
        nResult = true;
    }
```

```
    else {
        fDelta = pfaFan->vx1 * pfaFan->vy2 - pfaF
        fAlpha = ( dx * pfaFan->vy2 - dy * pfaFar
        fBeta = ( -dx * pfaFan->vy1 + dy * pfaFar
        if ( ( fAlpha >= 0.0f ) && ( fBeta >= 0.0
            ar = pfaFan->r + pcrCircle->r;
            if ( fDistSqr <= ar * ar ) {
                nResult = true;
            }
        }
        else {
            a = pfaFan->vx1 * pfaFan->vx1 + pfaFa
            b = -( pfaFan->vx1 * dx + pfaFan->vy1
            c = dx * dx + dy * dy - pcrCircle->r
            d = b * b - a * c;
            if ( d >= 0.0f ) {
                t = ( -b - sqrtf( d ) ) / a;
                if ( ( t >= 0.0f ) && ( t <= 1.0f
                    nResult = true;
                }
            }
            a = pfaFan->vx2 * pfaFan->vx2 + pfaFa
            b = -( pfaFan->vx2 * dx + pfaFan->vy2
            c = dx * dx + dy * dy - pcrCircle->r
            d = b * b - a * c;
            if ( d >= 0.0f ) {
                t = ( -b - sqrtf( d ) ) / a;
                if ( ( t >= 0.0f ) && ( t <= 1.0f
                    nResult = true;
                }
            }
        }
    }

    return nResult;
}
```

程序很长。这是因为扇形的碰撞检测需要分别处理很多不同的条件分支。虽然可能会有人觉得程序太长不好理解，但请耐下心来，让我们一起来逐步看一下。

首先看向函数传递的参数。传递的结构体中，F_CIRCLE 结构体表示圆形，通过圆心坐标及半径表示特定的圆。F_FAN 结构体表示扇形，结构体的成员乍看上去可能有点乱，因为有很多与碰撞检测无关的成员。与碰撞检测有关的部分，是一个点与两个向量，它们可以共同表示一个特定的扇形（参考图 3-4-2），其他的成员（角）可以暂时忽略。

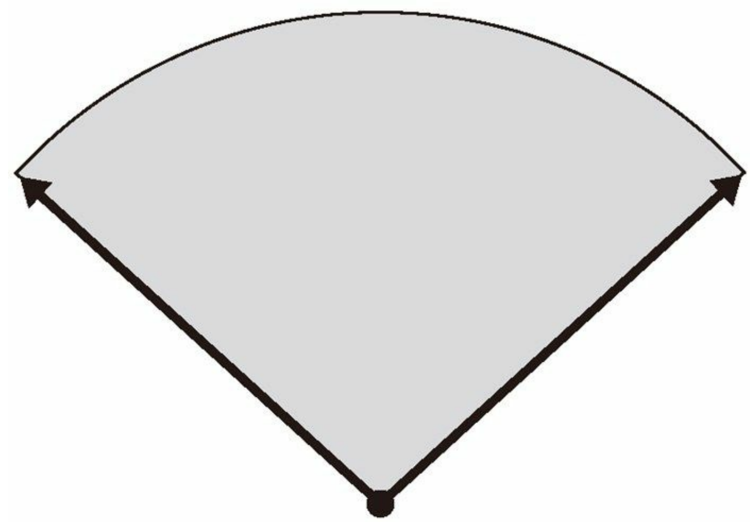


图 **3-4-2**　一个点与两个向量表示扇形

CheckHit 函数中，会对圆心坐标及半径表示的圆，与一点加两个向量表示的扇形进行碰撞检测。此时只用一个算式很难覆盖所有可能的碰撞条件，需要先根据不同的条件进行划分。至于如何划分，可能会有很多种思路，我们采用的划分方式如示例程序 CheckHit_4_1.cpp 所示，有以下几种情况。

条件 1. 扇形的圆心点，如果在圆形内部则表示碰撞（参考图 3-4-3 左）。

关于这一点，应该无需特别说明。下文将主要考虑扇形的圆心在圆外的情况。

条件 2. 圆的圆心坐标，如果在扇形内部则表示碰撞（参考图 3-4-3 右）。

即使扇形的圆心在圆外，只要圆的圆心坐标被扇形包含，就认为两者是碰撞的。

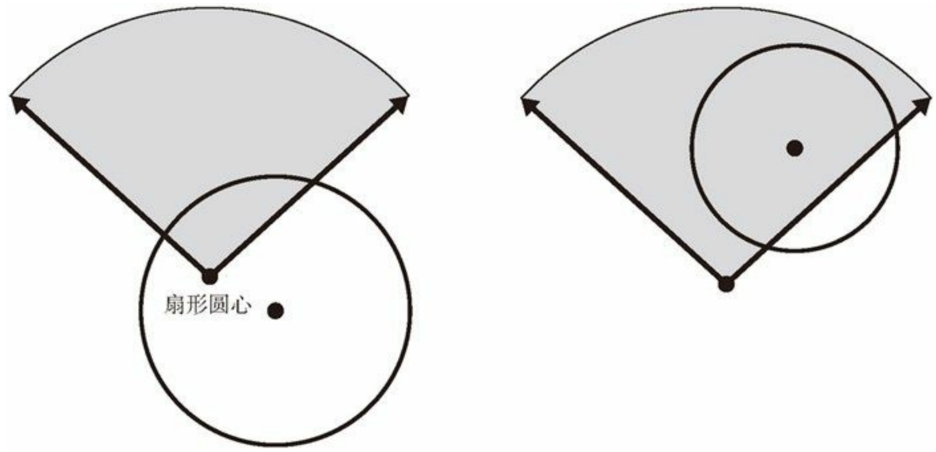


图 **3-4-3**　扇形与圆形碰撞时的位置关系（其一）

那么上面的条件是否已经覆盖了所有的情况呢？不，还有遗漏。即便两者的圆心坐标都没有被对方包含，图形的周边部分如果有交叠，那么两个图形仍然会碰撞。因此需要追加以下两个条件：

> 条件 2′. 圆形的圆心坐标在组成扇形的两个向量之间，并且圆形的圆心坐标到扇形的圆心坐标的距离，小于“圆形半径 + 扇形半径”时，认为两图形碰撞（参考图 3-4-4 左）。

本条件覆盖了条件 2 中圆的圆心坐标在扇形内的情况，并且还考虑到了圆在扇形内且扇形的弧与圆有交叠的特殊情况。除此以外的情况下扇形的弧与圆也有可能交叠，请参考下文中的条件 3。由于本条件已经完全覆盖了条件 2，因此条件 2 的检测可以省略。

> 条件 3. 扇形的两个向量所构成的外侧边缘线段中，只要其中任意一条与圆有交点就认为碰撞（参考图 3-4-4 右）。

本条件补充了条件 2′ 中遗漏的部分，即条件 2′ 以外的扇形的弧与圆有交叠的所有情况。

因为当圆的圆心坐标没有在扇形的两个向量之间时，如果圆与扇形仍然有交叠，则说明扇形边缘的线段与圆必有一个交点（或者也可以说圆与扇形的弧的交点不可能有两个）。

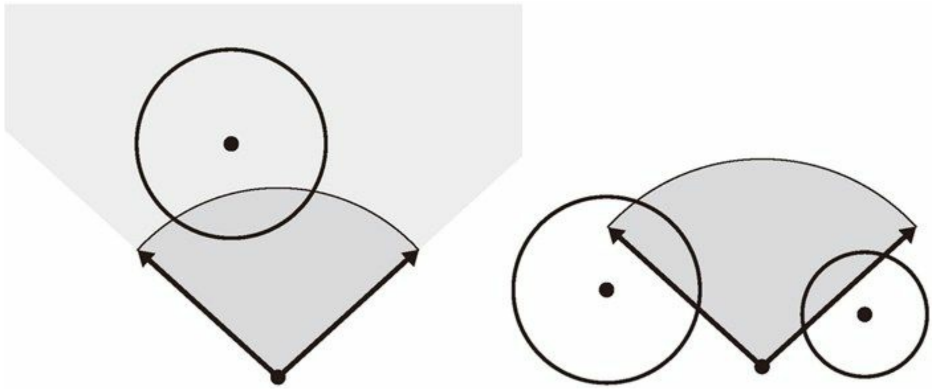


图 **3-4-4**　扇形与圆形碰撞时的位置关系（其二）

条件 1、2′、3 中，只要满足任意一个，都可以认为圆形与扇形有碰撞，下面将按程序的顺序逐条说明检测是如何进行的。

- 条件 **1.** 扇形圆心在圆形内部

  条件 1 是扇形圆心在圆形内部时，两图形碰撞。令圆的圆心坐标为 (x_C, y_C)、圆的半径为 r_c、扇形的圆心坐标为 (x_F, y_F)，根据勾股定理，有以下关系成立。

  $$(x_C - x_F)^2 + (y_C - y_F)^2 lt; r_C^2$$

  将其程序化，对应 CheckHit 函数中的以下部分。

```
    dx = pcrCircle->x - pfaFan->x; dy = pcrC
    fDistSqr = dx * dx + dy * dy;
    if ( fDistSqr < pcrCircle->r * pcrCircle
        nResult = true;
    }
```

对于条件 1，应该不需要特别说明。

- 条件 **2′ .** 圆的圆心坐标在扇形的向量之间

接下来是条件 2′：圆形的圆心坐标在组成扇形的两个向量之间，并且圆形的圆心坐标到扇形的圆心坐标的距离，小于“圆形半径 + 扇形半径”时，认为两图形碰撞。令扇形的两个向量分别为 v_1、v_2，圆的圆心坐标为 (x_C, y_C)（这个圆心坐标的位置向量表示为 C），那么圆的圆心在 v_1、v_2 之间这个条件可以表示为

$$C - F = \alpha v_1 + \beta v_2 \quad \alpha > 0 \text{ 且 } \beta > 0$$

上式中的 F 是扇形的圆心坐标 (x_F, y_F) 的位置向量。看到上面的等式，会很自然地联想到向量的定比分公式，因为定比分中的内分公式，正是用于计算向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 之间比例为 t:1-t 的位置向量的公式，假设内分点的位置向量为 $\boldsymbol{p}$，则有

$$\boldsymbol{p} = (1 - t)\boldsymbol{a} + t\boldsymbol{b} \quad (0 \le t \le 1)$$

而用 α 表示 (1-t)，用 β 表示 t，公式就可以转换为

$$\boldsymbol{p} = \alpha\boldsymbol{a} + \beta\boldsymbol{b} \quad (\alpha \ge 0 \text{ 且 } \beta \ge 0 \text{ 且 } \alpha + \beta = 1)$$

当 α 与 β 满足上面的条件时，向量 $\boldsymbol{p}$ 就是向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 的内分点。当 $\alpha + \beta$ 不等于 1，而 是等于常数 d 时，等式可变形为

$$\begin{aligned}\boldsymbol{p} &= \alpha\boldsymbol{a} + \beta\boldsymbol{b} \\ &= d(\alpha'\boldsymbol{a} + \beta'\boldsymbol{b})\end{aligned}$$

其中 $\alpha' + \beta' = 1$，向量 $\alpha'\boldsymbol{a} + \beta'\boldsymbol{b}$ 表示向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 的内分点。此时原向量 $\boldsymbol{p}$ 就等于这个内分点乘以常数 d，那么如果在 $d > 0$ 的范围内移动 d，向量 $\boldsymbol{p}$ 就会变成一个从扇形的圆心坐标开始、通过内分点的射线上的点（参考图 3-4-5）。

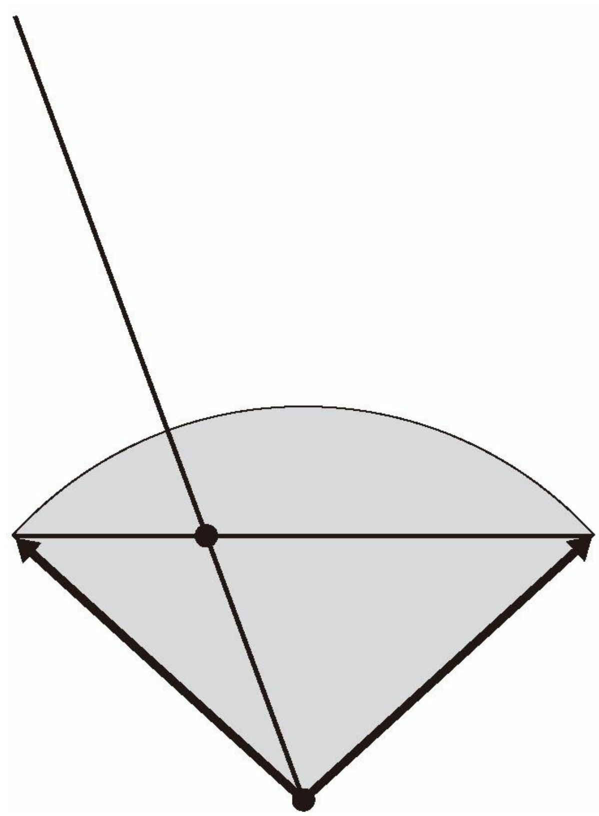


图 **3-4-5**　通过内分点的射线

对 α 与 β 进行各种取值（在正数范围内），这条射线能通过的所有范围正是向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 之间的区域。而由于 $\alpha > 0$ 且 $\beta > 0$ 时 $\alpha + \beta > 0$，且 $C - F$ 是一条由扇形圆心指向圆形圆心的向量，因此“$C - F = \alpha v_1 + \beta v_1$ 时 $\alpha > 0$ 且 $\beta > 0$”是 C 在向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 之间的条件（参考图 3-4-6）。

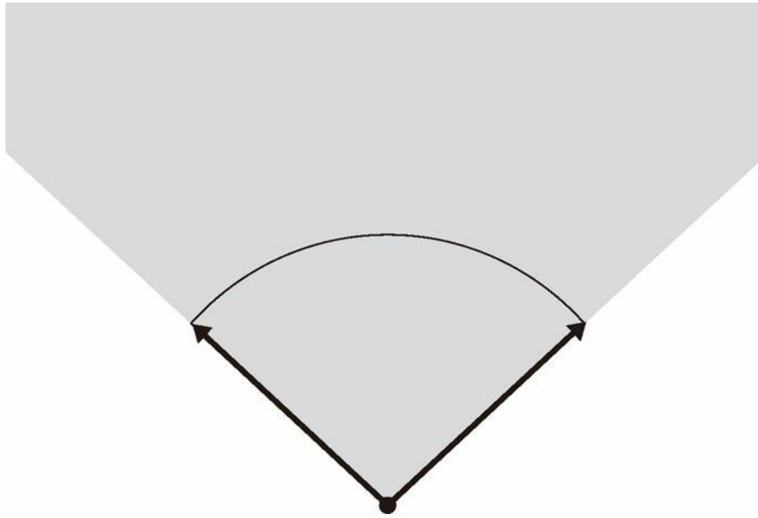


图 **3-4-6**　向量 *a* 与向量 *b* 之间的范围

上面的说明可能有些长，不知道读者朋友是否都已经明白了。将上面的过程代码化，就得到了 CheckHit 函数的如下部分。

```
        fDelta = pfaFan->vx1 * pfaFan->vy2 -
        fAlpha = ( dx * pfaFan->vy2 - dy * p
        fBeta = ( -dx * pfaFan->vy1 + dy * p
        if ( ( fAlpha >= 0.0f ) && ( fBeta >
            ar = pfaFan->r + pcrCircle->r;
            if ( fDistSqr <= ar * ar ) {
                nResult = true;
            }
        }
```

在这部分代码中，首先计算等式

$$\boldsymbol{C}-\boldsymbol{F}=\alpha v_1+\beta v_2$$

中的 α 与 β。若 $\boldsymbol{C}=(x_C, y_C)$、$\boldsymbol{F}=(x_F, y_F)$、$\boldsymbol{v}_1=(v_{1x}, v_{1y})$、$\boldsymbol{v}_2=(v_{2x}, v_{2y})$，上式就可以表示为

$$\begin{pmatrix} x_C - x_F \\ y_C - y_F \end{pmatrix} = \alpha \begin{pmatrix} v_{1x} \\ v_{1y} \end{pmatrix} + \beta \begin{pmatrix} v_{2x} \\ v_{2x} \end{pmatrix}$$

这个等式是将方程组

$$\begin{cases} x_C - x_F = \alpha v_{1x} + \beta v_{2x} \\ y_C - y_F = \alpha v_{1y} + \beta v_{2y} \end{cases}$$

使用列向量表示得到的。而如果使用矩阵，等式还可以写成

$$\begin{pmatrix} x_C - x_F \\ y_C - y_F \end{pmatrix} = \begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

如果觉得上式不好理解，可以自己实际计算一下矩阵的乘法，看看是否与前式的方程组一致。

对上面含有矩阵的等式，乘以矩阵 $\begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix}$ 的逆矩阵，让我们看看可以得到什么。

$$\begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix}^{-1} \begin{pmatrix} x_C - x_F \\ y_C - y_F \end{pmatrix} = \begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix}^{-1} \begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

由于

$$\begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix}^{-1} \begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix} = \boldsymbol{E}$$

单位矩阵会被消掉，因此左右对调一下可以得到

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix}^{-1} \begin{pmatrix} x_C - x_F \\ y_C - y_F \end{pmatrix}$$

由于 $\begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix}^{-1}$ 是 2×2 矩阵的逆矩阵，所以用矩阵的计算公式

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

简单求解可有

$$\begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix}^{-1} = \frac{1}{v_{1x}v_{2y}-v_{2x}v_{1y}}\begin{pmatrix} v_{2y} & -v_{2x} \\ -v_{1y} & v_{1x} \end{pmatrix}$$

然后得到

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \frac{1}{v_{1x}v_{2y}-v_{2x}v_{1y}}\begin{pmatrix} v_{2y} & -v_{2x} \\ -v_{1y} & v_{1x} \end{pmatrix}\begin{pmatrix} x_C - x_F \\ y_C - y_F \end{pmatrix}$$

这里将系数的分母 $v_{1x}v_{2y}$-$v_{2x}v_{1y}$ 表示为 Δ，则有

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \frac{1}{\Delta}\begin{pmatrix} v_{2y} & -v_{2x} \\ -v_{1y} & v_{1x} \end{pmatrix}\begin{pmatrix} x_C - x_F \\ y_C - y_F \end{pmatrix}$$

则 α 与 β 分别为

$$\begin{cases} \alpha = \dfrac{1}{\Delta}\{v_{2y}(x_C - x_F) - v_{2x}(y_C - y_F)\} \\ \beta = \dfrac{1}{\Delta}\{-v_{1y}(x_C - x_F) + v_{1x}(y_C - y_F)\} \end{cases}$$

这对应 CheckHit 函数中的如下部分。

```
        fDelta = pfaFan->vx1 * pfaFan->vy2 -
        fAlpha = (  dx * pfaFan->vy2 - dy *
        fBeta = (  -dx * pfaFan->vy1 + dy *
```

这部分处理的是“条件 2′. 圆形的圆心坐标在组成扇形的两个向量之间，并且圆形的圆心坐标到扇形的圆心坐标的距离，小于“圆形半径 + 扇形半径”时，认为两图形碰撞”。由于圆形的圆心坐标

在扇形的两个向量之间的附加条件是 $a > 0$ 且 $b > 0$，并且还需要检查圆形的圆心坐标到扇形的圆心坐标的距离，因此程序中通过以下部分

```
        if ( ( fAlpha >= 0.0f ) && ( fBeta >
            ar = pfaFan->r + pcrCircle->r;
            if ( fDistSqr <= ar * ar ) {
                nResult = true;
            }
        }
```

进行了检查，请注意其中变量 fDistSqr 的值在处理条件 1 时，已经被设置为了圆的圆心坐标到扇形的圆心坐标的距离的平方。

另外，从一点是否在两向量之间的检测以及 Δ、α 、β 的算式中，可能已经有人注意到了，这其实就等同于计算两向量的外积。检测一点是否在两向量之间，稍加注意的话是完全可以通过外积的方法进行检测的，并且算式可能会更加简单，但是一般来说，通过方程组求解的检测方式，可以适用于更多的情况，因此这里采用了方程组求解的方式。

- 条件 **3.** 扇形的两个向量构成的线段中任意一条与圆有交点时

至此虽然已经做了很长的讲解，不过我们还剩下最后一个条件没有说明，即“条件 3. 扇形的两个向量所构成的外侧边缘线段中，只要其中任意一条与圆有交点就认为碰撞”，这个条件涉及了圆与线段 的交点判断。请注意这不是圆与直线 的交点判断。如果是判断圆与直线，只要联立圆的方程式与直线的方程式得到二元方程组，然后求解得到交点的判别式 D ，并看 D 的符号就可以

知道是否相交。而这里与直线不同的是，线段有两个端点，因此判断线段与圆是否相交需要更加注意。

下面尝试计算一下圆心坐标为 $\boldsymbol{C}(x_C, y_C)$、半径为 r_c 的圆，与起点为 $\boldsymbol{F}(x_F, y_F)$、终点为 $\boldsymbol{F}+\boldsymbol{v}_1$ 的线段的交点。首先，圆心坐标为 $\boldsymbol{C}(x_C, y_C)$、半径为 r_c 的圆的方程式为

$$(x-x_C)^2+(y-y_C)^2=r_C^2$$

使用向量将起点为 $\boldsymbol{F}(x_F, y_F)$、终点为 $\boldsymbol{F}+\boldsymbol{v}_1$ 的线段表示为

$$\boldsymbol{P}=\boldsymbol{v}_1\boldsymbol{t}+\boldsymbol{F} \quad (0 \leq t \leq 1)$$

可将其分解为

$$\begin{cases} x=v_{1x}t+x_F \\ y=v_{1y}t+y_F \end{cases} \quad (0 \leq t \leq 1)$$

然后将线段等式代入圆的方程式，可以得到

$$(v_{1x}t+x_F-x_C)^2+(v_{1y}t+y_F-y_C)^2=r_c^2$$

接下来有点繁琐，我们要将括号的平方部分展开

$$v_{1x}^2t^2+2(x_F-x_C)v_{1x}t+(x_F-x_C)^2+v_{1y}^2t^2+2(y_F-y_C)v_{1y}t+(y_F-y_C)^2=r_c^2$$

合并整理一下有相同次数的 t，并令等式右边为零，有

$$(v_{1x}^2+v_{1y}^2)t^2+2\{(x_F-x_C)v_{1x}+(y_F-y_C)v_{1y}\}t+(x_F-x_C)^2+(y_F-y_C)^2-r_c^2=0$$

求解上式可以参考二次方程

$$ax^2+2bx+c=0$$

的解

$$x = \frac{-b \pm \sqrt{b^2 - ac}}{a}$$

将要求的变量改为 t，这时上式中的各系数分别为

$$a = v_{1x}^2 + v_{1y}^2$$
$$b = -\{(x_C - x_F)v_{1x} + (y_C - y_F)v_{1y}\}$$
$$c = (x_C - x_F)^2 + (y_C - y_F)^2 - r_c^2$$

而在程序中，将 $(x_F - x_C)$ 置换为了 $-(x_C - x_F)$，因为通过这个小技巧，可以重复利用上面程序中已经计算出来的 $(x_C - x_F)$。

最后将其代入上面求解的公式，就得到了 CheckHit 函数中的如下部分。

```
        a = pfaFan->vx1 * pfaFan->vx1 +
        b = -( pfaFan->vx1 * dx + pfaFar
        c = dx * dx + dy * dy - pcrCircl
        d = b * b - a * c;
        if ( d >= 0.0f ) {
            t = ( -b - sqrtf( d ) ) / a;
            if ( ( t >= 0.0f ) && ( t <=
                nResult = true;
            }
        }
```

其中

```
        d = b * b - a * c;
```

这一行计算了变量 d，并检测了 d 的符号。这里的变量 d 其实就是二次方程的判别式 D，也就是 解的公式中根号（$\sqrt{\quad}$ ）内的部分。当判别式为负时，不单是线段，包含线段的整条直线与圆都没有交点，因此需要注意将负值的条件分支排除在外，不作为交点的判断条件。

当判别式大于 0 时，需要计算交点所对应的 t，当 t 在范围 $0 \le t \le 1$ 内时，线段与圆有交点。但在程序中，对于公式求得的两个解，这里只对其中一个进行了范围检测。具体来说就是正负号的部分取负得到的

$$t = \frac{-b - \sqrt{b^2 - ac}}{a}$$

这个解。程序中只检测了该解的取值范围，这是为什么呢？其实这是由于正负号中取负的解是两个解中较小的一个。因为在实数范围内开根号后的数必定大于零，那么减去这样一个正数肯定要比加上这个正数的解要小。

之所以只对较小的解进行检测，是基于以下原因。当线段与圆有交点时，有 3 种可能：① 仅较大的解与线段有交点；②仅较小的解与线段有交点；③两个解都与线段有交点。但是仅有较大的解与线段有交点时，扇形的圆心必定在圆内（参考图 3-4-3 左）。而这种情况已经在上文中的条件 1 中进行过判断，既然程序已经运行到了此处，就说明扇形的圆心是不可能在圆内的，也就是说这时不可能仅有较大的解与线段有交点，因此我们只需要检测较小的解就可以了（参考图 3-4-4）。这就是为什么此处的程序只对正负号中取负的解进行了检查。

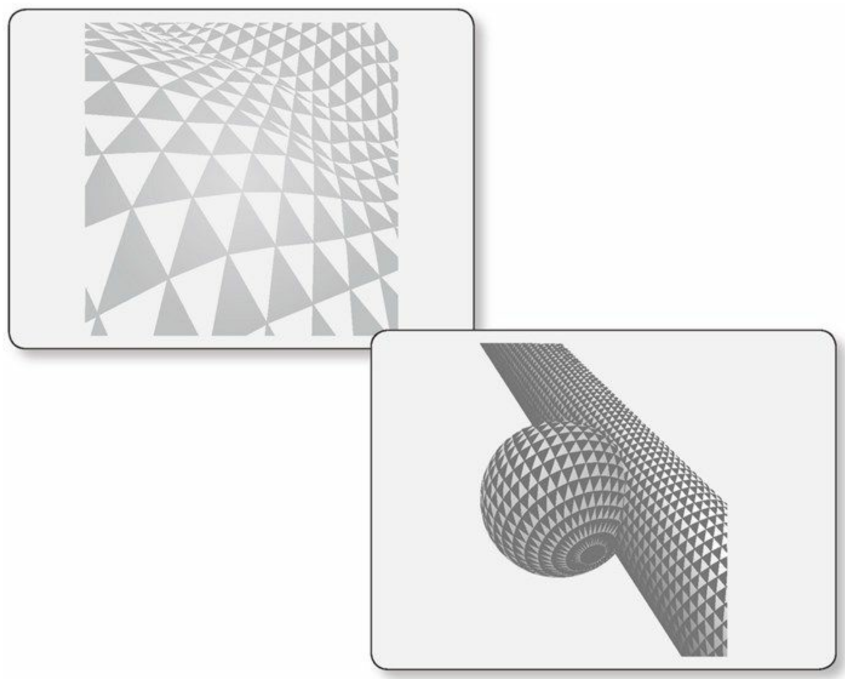
经过了相当长的讲解，总算把圆形与扇形的碰撞检测介绍完了。程序中接下来还对扇形的向量 v_2 所代表的线段与圆是否相交进行了检测，原理与向量 v_1 与圆的相交检测是一样的，这里就不再赘述了。

在上述的圆形与扇形的碰撞检测中，一方面为了易于理解，另一方面也想让本书涉及更多的数学话题，程序中有些地方在速度及效率上做了一定的妥协。比如在实际项目中，大多数情况下都会优先检测两物体绝对不可能碰撞的条件，即对圆的圆心坐标与扇形的圆心坐标的距离是否大于“圆的半径 + 扇形的半径”进行检测，并将不可能碰撞的情况排除。比如在挥动剑时就可以使用这种检测方式，因为物体绝大多数情况下都是在剑的所及范围之外的。这种情况下程序要如何编写，读者们可以自己尝试着实现一下。

3.5 [进阶] 3D 的碰撞检测



2D、**3D**、维度扩展



本章所介绍的碰撞检测都是基 于 **2D** 的，这是为了让读者先掌握碰撞检测的基础，因此刻意回避了一 些复杂的算法，并且优先了程序的易读性。而在本章的最后部分，就让我们一起来了解一下基于 **3D** 的碰撞检测的基本知识吧。

本小节将简单说明相对于 2D 的碰撞检测，3D 碰撞检测究竟有多复杂。由于 3D 碰撞检测相对于

2D 来说增加了一个维度，即表示坐标的变量从二维的 (x, y)，变为了三维的 (x, y, z)。因此根据检测方法的不同，碰撞检测的复杂程度会有很大差别。

比如，将我们在 3.1 节介绍的与坐标轴平行的长方形物体间的碰撞检测，扩展为与坐标轴平行的长方体物体间的碰撞检测，此时虽然从 2D 扩展为了 3D，但是复杂度其实并没有增加很多。实际来试试吧。首先，如果基于 2D 的与坐标轴平行的长方形之间有如下关系（此处为了方便数学表达，与 3.1 节的说明略有不同）。

$$|x_1 - x_2| \leq \frac{w_1 + w_2}{2}$$

且

$$|y_1 - y_2| \leq \frac{h_1 + h_2}{2}$$

则可认为两物体碰撞。此处的 (x_1, y_1) 与 $(x_2, y$
$)$ 分别代表长方形 1 与长方形 2 的中心坐标，w_1 与 w_2 分别代表长方形 1 与长方形 2 的长度，h_1 与 h_2 分别代表长方形 1 与长方形 2 的宽度（参考图 3-5-1）。

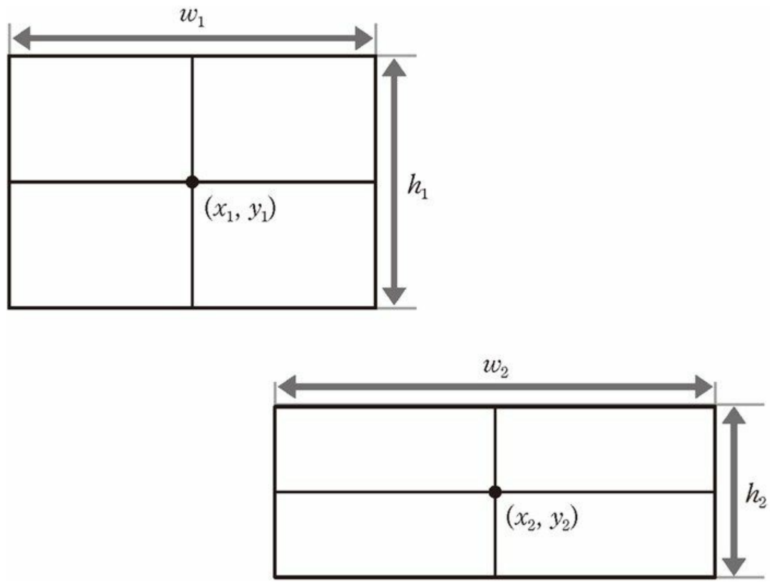


图 **3-5-1**　要检测的物体为长方形

将其扩展到 3D 的话，则当

$$|x_1 - x_2| \leq \frac{w_1 + w_2}{2}$$

且

$$|y_1 - y_2| \leq \frac{h_1 + h_2}{2}$$

且

$$|z_1 - z_2| \leq \frac{d_1 + d_2}{2}$$

都满足时，则两物体碰撞。此处 (x_1, y_1, z_1) 与 $(x_2, y_2$, z2) 分别代表长方体 1 与长方体 2 的中心坐标，d_1 与 d_2 分别代表长方体 1 与长方体 2 的高度，与 2D 相比，仅附加了 *z* 坐标的判断条件，等式的复杂程度就变为了原来的 1.5 倍（与维度数相比）。

另一个类似的例子是 3.2 节介绍的圆形的碰撞检

测，如果将其简单地扩展为 3D 的话，就变成了球体之间的碰撞检测。也来实际尝试一下吧。首先，当满足

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 \leq (r_1 + r_2)^2$$

时，则认为两圆形物体碰撞。其中 (x_1, y_1) 与 (x_2, y_2) 分别代表圆 1 与圆 2 的圆心坐标，r_1 和 r 分别代表圆 1 与圆 2 的半径。将其扩展到球体间的碰撞检测，则是当满足

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 \leq (r_1 + r_2)^2$$

时，两球体碰撞。这里的 (x_1, y_1, z_1) 与 $(x_2, y$ z2) 分别代表球体 1 与球体 2 的中心坐标。这个例子与 2D 的情况相比，只是附加了 z 项，等式的复杂度没有增加。

通过上面两个例子，大家可能会认为即使碰撞检测变为 3D，其复杂程度也没有增加很多。但这并不适用于所有情况，因为根据物体运动所要求的自由度、物体形状的变化，以及要处理的变量增多时所造成的等式的复杂化等，3D 碰撞检测的复杂程度是会变化的。

比如对正方形物体的碰撞检测，在大多数情况下，2D 中只需要考虑各边与 *x* 、*y* 轴平行的情况，3D 中只需要考虑立方体各面与 *x* 、*y* 、*z* 轴平行的情况（即上文中刚刚讲过的情况）。因为在 2D 中，角色本身需要旋转的情况并不多见，而在 3D 中大多数情况下角色本身或者角色的构成部件都会转动。另外当立方体各面与轴都不平行时，其碰撞检测会变得复杂很多（其实即便是 2D 的长方形，在旋转时进行碰撞检测也是很麻烦的）。

在上面的圆形与球体的例子中，我们将 2D 的圆形扩展为了 3D 的球体，而换一种思考方式，由

圆形扩展为圆柱体或圆锥体都是完全有可能的。例如当圆形扩展为圆柱体时，圆柱体之间的碰撞检测就要比球体之间的碰撞检测的计算复杂得多。

另外，当 2D 的直线扩展到 3D 时，由于本来是坐标空间低一个维度、可以用一次方程表示的图形，因此一般就会将直线扩展为平面。在 2D 中，一条通过任意个点的直线，只需要两个点就可以确定，而在 3D 中，一个通过任意个点的平面，则必须有三个点才能确定。这就好比想要稳定地架起一根筷子，只需要在筷子下放两个支点就可以了，而想要架起一个杯垫，就必须在垫子下面放至少三个支点。由两点决定的直线，只需要联立有两个未知数的二元方程就可以求解，通过矩阵求解也只需要计算一个 2×2 矩阵的逆矩阵，复杂度与方程组一样。而对于一个由三个点才能确定的平面来说，就需要求解有三个未知数的三元联合方程，或者计算有着同样复杂度的 3×3 矩阵的逆矩阵。众所周知，求 2×2 矩阵的逆矩阵是比较简单的，而求解 3×3 矩阵的逆矩阵就要复杂得多了。因此程序中一般不会采用三元联合方程的形式来计算由三个点确定的平面，而是将三个点转换为一个点加两个向量，通过计算通过一个点且与两个向量平行的平面来求解。

想必上面的例子已经可以说明，仅仅将 2D 中的解法简单地扩展到 3D，反而会让问题变得复杂，因此针对 3D 往往采用 3D 特有的解法来处理问题。这也是相比 2D 来说 3D 的碰撞检测更加复杂的原因。此外，如果想向别人讲解碰撞检测的数学知识，或者想自学新的碰撞检测，2D 的情况下只要在纸上画出图形就可以进行说明，而如果是 3D，想要在纸上清楚地表示 3D 空间则基本不太可能，因此无论是从说明还是从学习的角度来讲，3D 的情况都更加困难。

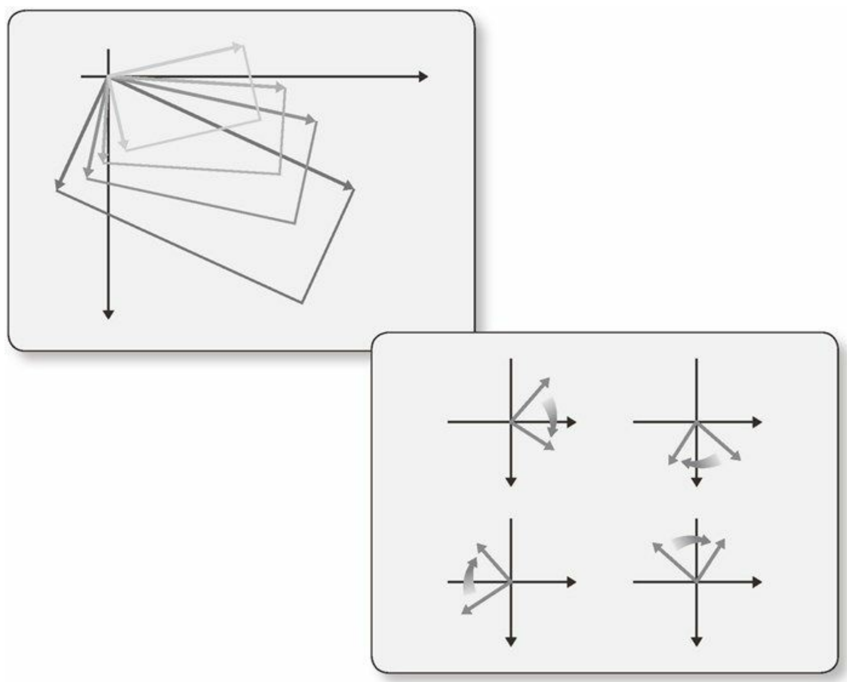
可想而知，在游戏开发中，从 2D 过渡到 3D 时，碰撞检测所需要的数学公式及算法会更加复杂，并且不容易理解，这就造成了 3D 开发的门槛较高。不过门槛高也有好处，就是能从事这类开发的人才比较稀缺。本书中并没有涉及 3D 的碰撞检测，如果读者立志要成为游戏开发者，并且希望有较高身价的话，不妨以本书的 2D 碰撞检测为基础，去挑战更难的 3D 碰撞检测吧。

第 **4** 章　光线的制作

4.1　让物体向任意方向旋转（含缩放效果）

Key Word　旋转、基向量、向量加法、向量减法

RANK/normal



在第 **1** 章 中，我们已经介绍了能改变角色等物体位置的“运动”。从本章开始，我们将讲解可以改变物体方向的“旋转”。首先来看以物 体自身的一点为中心旋转的情况。

本小节将介绍如何让物体旋转并朝向指定的方

向。假设有一张光线的图片素材，通过本小节的学习就可以实现将光线投射到某个特定的方向。而为了让数学部分更加简单，我们只考虑物体在原点 O 时的旋转。



图 **4-1-1**　通过物体旋转实现的光线显示的程序

提到物体的旋转，具备一定程度的数学知识的朋友可能首先都会想：使用旋转矩阵不就可以实现了吗？的确，旋转矩阵是游戏中常用的方法，在物体面向的角度确定时非常有效。然而目前我们是想通过倾斜一张光线素材图片来实现光线的投射，此时旋转矩阵就不太适用了。因为游戏中在投射光线等物体时，其投射方向的角度往往是未知的，一般只能知道光线的起点及终点，这在 3D 游戏中尤为多见。

在光线的起点及终点都确定的情况下，如果采用旋转矩阵来倾斜物体，首先就需要连接起点与终点得到一条直线，然后求得这条直线与 x 轴的夹角，通过夹角就可以得到一个旋转矩阵并控制物体的旋转。这种方法确实可以实现让物体朝向指定的方向，但是计算过程中会存在很多浪费。下面从数学角度来说明原因。通过倾斜的直线先求

得角度，再通过角度确定旋转矩阵并控制物体旋转这一操作，从本质上来讲等于将普通的坐标系先转换到包含角度的特殊坐标系（例如极坐标这种坐标系），然后再重新转换回普通坐标系。然而如果只是为了让物体旋转，这些坐标系间的转换其实都是多余的。而且从工程学的角度来看，求直线与 *x* 轴的夹角会用到反三角函数（如 C 语言中的 atan2 等），反三角函数的计算更花时间，应该尽量避免使用。基于这些原因，我们不会通过旋转矩阵（或反三角函数等）的方式来旋转物体。

不使用旋转矩阵，具体要如何操作呢？我们会用到基变换 这一方法。基变换就是对基向量进行的变换，而所谓基向量，就是决定 *x* 轴、*y* 轴方向与放大比率的向量。比如在普通的坐标系中，*x* 方向的基向量为 $\boldsymbol{i}=(1, 0)$，*y* 方向的基向量为 $\boldsymbol{j} = (0, 1)$。将这两个基向量组合，就可以将任意坐标 (x , y) 表示为 $x\boldsymbol{i} +y\boldsymbol{j}$ 的形式。这样一来，即使不更改 *x* 、*y* ，而只是将 $\boldsymbol{i}$ 、 $\boldsymbol{j}$ 旋转指向特定的方向，也可以实现物体的旋转（参考图 4-1-2）。

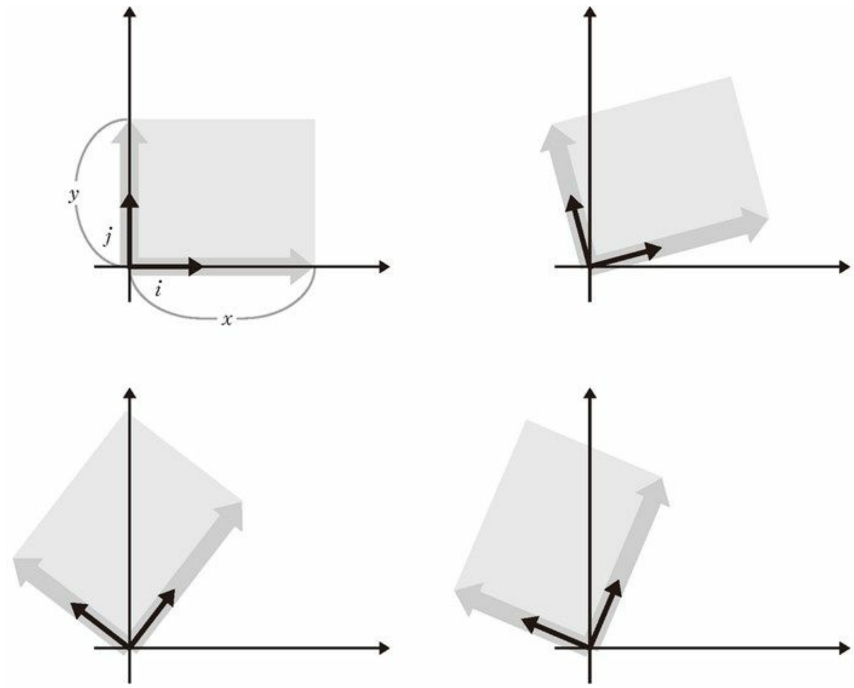


图 **4-1-2**　通过 *i* 与 *j* 来实现物体的旋转

但是，如果保持 ***i*** 、 ***j*** 的位置关系不变（图中 ***j*** 位于 ***i*** 左方向经过一个直角的位置），物体是无法进行旋转的。此时如果改变 *x* 、*y* ，就会破坏 ***i*** 与 ***j*** 的正交关系，这种变换称为切变 （shear conversion），看起来有点像压扁了一个纸箱（参考图 4-1-3）。

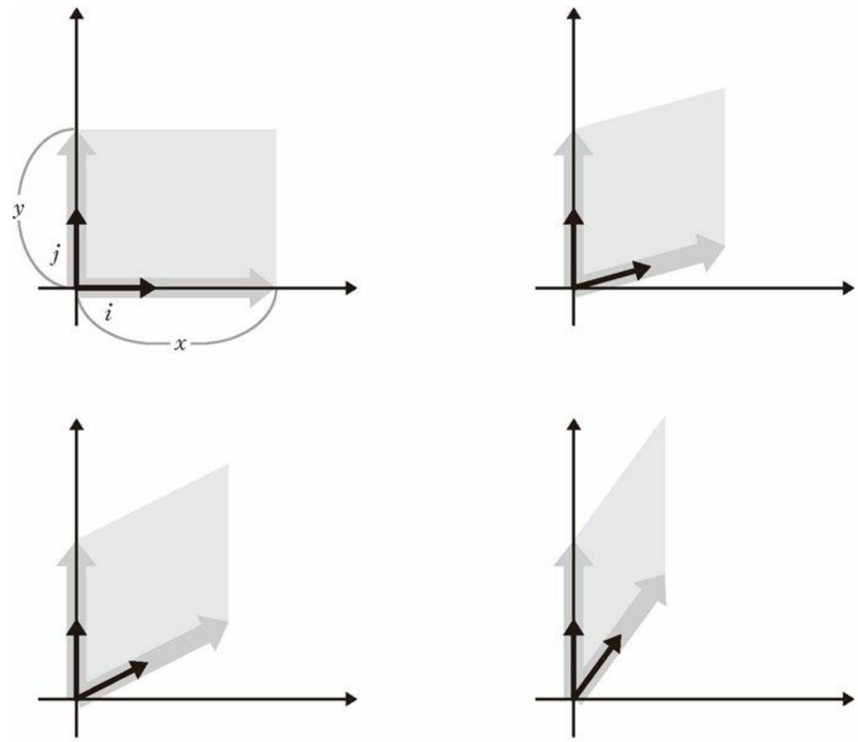


图 **4-1-3**　切变

这里，我们假设基向量 ***i*** 指向任意方向，并将 ***i*** 沿顺时针方向旋转 $\frac{\pi}{2}$（90 度）得到 ***j*** 。而在上例中，之所以将 ***i*** 逆时针旋转得到 ***j*** ，是因为在计算机中，*y* 轴是指向下方的。

假设这样得到的一对新的基向量为 ***i*′** 与 ***j*′** 。然后进行向新坐标系的坐标变换，坐标为 (*x* ,*y*) 的点 ***P*** ，即 ***P*** =*x**i*** +*y**j*** 这个点，就移动到了点 ***P*′** =*x**i*** ′ +*y**j*** ′ 。可将其用分量表示为

$$\begin{pmatrix} P'_x \\ P'_y \end{pmatrix} = x \begin{pmatrix} i'_x \\ i'_y \end{pmatrix} + \begin{pmatrix} j'_x \\ j'_y \end{pmatrix}$$

用矩阵可表示为

$$\begin{pmatrix} P'_x \\ P'_y \end{pmatrix} = \begin{pmatrix} i'_x & j'_x \\ i'_y & j'_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

也就是说，只要将新的基向量 ***i′*** 、 ***j′*** 的分量表示为矩阵形式，就可以得到从一般坐标系到新坐标系的变换矩阵。这是一个非常方便的数学结论，最好可以背下来（普通的旋转矩阵就是通过这种方式得到的矩阵的一个特例）。

> **POINT**　要得到从普通坐标系到新坐标系的变换矩阵，只需要将新的基向量的分量表示为矩阵形式就可以了。

接下来就让我们使用这个方便的矩阵，来实际将物体旋转一下。示例程序 Ray_1_1.cpp 就是一个简单的旋转物体的程序。以激光图片的左上角为原点，使图片的右上角跟随鼠标指针旋转以让整张图片转动。这个程序不仅实现了物体的旋转，还可以改变物体的大小，即具备放大缩小功能。为了便于说明，算法做了一定的简化。在这个同时具备旋转及放大缩小功能的程序 Ray_1_1.cpp 中，我们只需要看最重要的部分，即 MoveCharacter 函数中的以下部分（代码清单 4-1-1）。

代码清单 **4-1-1**　具备旋转及放大缩小功能的 **MoveCharacter** 函数中的主要部分（**Ray_1_1.cpp** 代码片段）

```
    v2Forward.x = ( float )CursorPos.x;
    v2Forward.y = ( float )CursorPos.y;
    v2Side.x = -v2Forward.y;
    v2Side.y =  v2Forward.x;

    v2Pos1.x = 0.0f;                       v2P
    v2Pos2.x = ( float )CursorPos.x;       v2P
    v2Pos3.x =              + v2Side.x;    v2P
    v2Pos4.x = CursorPos.x + v2Side.x;     v2P
```

其中 v2Forward 是一个代表 2D 向量的结构体，包含了原点到鼠标指针的向量，即上文中新的基向量 $\boldsymbol{i}'$ 。v2Side 结构体中包含了新的基向量 $\boldsymbol{j}'$ ，程序中的

```
    v2Side.x = -v2Forward.y;
    v2Side.y =  v2Forward.x;
```

这两行将基向量 $\boldsymbol{i}'$ 的 x 分量与 y 分量进行了互换，并将 x 分量（原 $\boldsymbol{i}'$ 的 y 分量）取负，得到了 $\boldsymbol{j}'$ 。为什么要这样变换呢？因为 $\boldsymbol{j}'$ 是将 $\boldsymbol{i}'$ 顺时针旋转 $\frac{\pi}{2}$ （90 度）得到的，此时 $\boldsymbol{i}'$ 与 $\boldsymbol{j}'$ 的位置关系应当为正交，而上式代表的就是正交的旋转变换结果。下面就来验证一下吧！将向量旋转 $\frac{\pi}{2}$ （90 度）的矩阵可变换为

$$\begin{pmatrix} \cos\frac{\pi}{2} & -\sin\frac{\pi}{2} \\ \sin\frac{\pi}{2} & \cos\frac{\pi}{2} \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

使用这个矩阵，假设将 $\boldsymbol{i}'$ 旋转 $\frac{\pi}{2}$ （90 度）得到 $\boldsymbol{j}'$ ，就有如下关系成立。

$$\begin{pmatrix} j'_x \\ j'_y \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i'_x \\ i'_y \end{pmatrix}$$

$$\therefore \begin{cases} j'_x = -i'_y \\ j'_y = i'_x \end{cases}$$

这与程序中的下列语句一致。

```
    v2Side.x = -v2Forward.y;
    v2Side.y =  v2Forward.x;
```

请注意在一般的教科书中，坐标系都以 y 轴上方为正，旋转 $\frac{\pi}{2}$ 是沿逆时针方向的。而在计算机中，坐标系以 y 轴下方为正，因此是顺时针旋转 $\frac{\pi}{2}$。

POINT　计算机中的坐标系以 y 轴下方为正，因此需要注意旋转方向与通常情况是相反的。

- 对四边形的四个角的坐标进行变换

既然已经得到了 $\boldsymbol{i}'$ 与 $\boldsymbol{j}'$ 这两个基向量，接下来就让我们使用它们对四边形的四个角的坐标进行变换。假设上例中变换前的四个角分别为：左上角 (0, 0)、右上角 (1, 0)、左下角 (0, 1)、右下角 (1, 1)（参考图 4-1-4）。

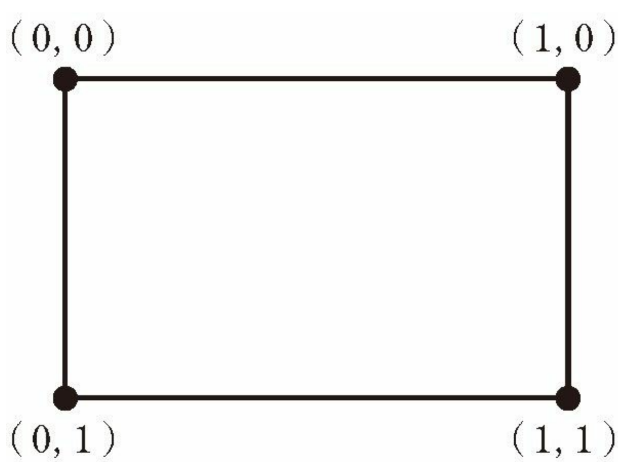


图 **4-1-4**　四边形的四个角的坐标

如果按照这个坐标直接绘制的话，只能得到一个几乎看不到的 1 像素大小的图形，将其

从原点拉伸至鼠标指针的位置，才能得到一个正常的四边形。此时使用上例的矩阵

$$\begin{pmatrix} i'_x & j'_x \\ i'_y & j'_y \end{pmatrix}$$

对四边形的四个角进行变换，让我们来看看会得到什么结果。

① 左上角

$$\begin{pmatrix} i'_x & j'_x \\ i'_y & j'_y \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$
$$= \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

② 右上角

$$\begin{pmatrix} i'_x & j'_x \\ i'_y & j'_y \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
$$= \begin{pmatrix} i'_x \\ i'_y \end{pmatrix}$$

③左下角

$$\begin{pmatrix} i'_x & j'_x \\ i'_y & j'_y \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$
$$= \begin{pmatrix} j'_x \\ j'_y \end{pmatrix}$$

④右下角

$$\begin{pmatrix} i'_x & j'_x \\ i'_y & j'_y \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$
$$= \begin{pmatrix} i'_x + j'_x \\ i'_y + j'_y \end{pmatrix}$$

在示例程序 Ray_1_1.cpp 中，上面的变换体

现为

```
v2Pos1.x = 0.0f;
v2Pos2.x = ( float )CursorPos.x;
v2Pos3.x =              + v2Side.x;
v2Pos4.x = CursorPos.x + v2Side.x;
```

其中 v2Pos1、v2Pos2、v2Pos3、v2Pos4 分别代表图形的左上角、右上角、左下角、右下角的坐标。可以看到，当程序中的基向量 ***i′*** 为 v2Forward（=CursorPos），基向量 ***j′*** 为 v2Side 时，矩阵计算的结果与程序计算的结果是一致的。其实上面的算式，除了可以看作是通过矩阵进行变换外，同时也是用向量 ***i′*** 、 ***j′*** 及其加法运算来表示图形的四个角的坐标（参考图 4-1-5）。

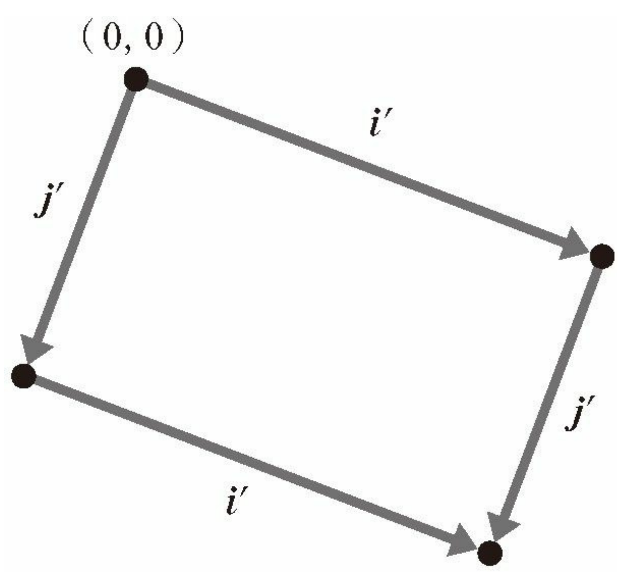


图 **4-1-5**　用向量 ***i′*** 、***j′*** 及其加法运算来表示四个角

- 以任意一点为中心旋转

接下来我们尝试不以图形的四个角中的一点为原点，而以任意一点为中心进行旋转。具体实现请参考示例程序 Ray_1_1a.cpp。程序中的重点是 MoveCharacter 函数中的以下部分（代码 清单 4-1-2）。

代码清单 **4-1-2**　使图形以任意一点为中心旋转的 **MoveCharacter** 函数的主要部分（**Ray_1_1a.cpp** 代码片段）

```
    v2Forward.x = ( float )CursorPos.x
    v2Forward.y = ( float )CursorPos.y
    v2Side.x = -v2Forward.y;
    v2Side.y = v2Forward.x;

    v2Pos1.x = ( float )OriginPos.x;
    v2Pos2.x = ( float )CursorPos.x;
    v2Pos3.x = OriginPos.x + v2Side.x;
    v2Pos4.x = CursorPos.x + v2Side.x;
```

下面让我们与前文中的 Ray_1_1.cpp 比较着来看。首先，计算代表向量 ***i′*** 的 v2Forward 时，坐标之间变成了减法运算。其中 CursorPos 代表鼠标指针的位置，OriginPos 则代表原 Ray_1_1.cpp 中的原点，即图形的四个角中的一个角的坐标。这两个坐标之间进行减法，等同于向量之间进行减法，得到的是从坐标 OriginPos 指向 CursorPos 的向量（参考图 4-1-6）。

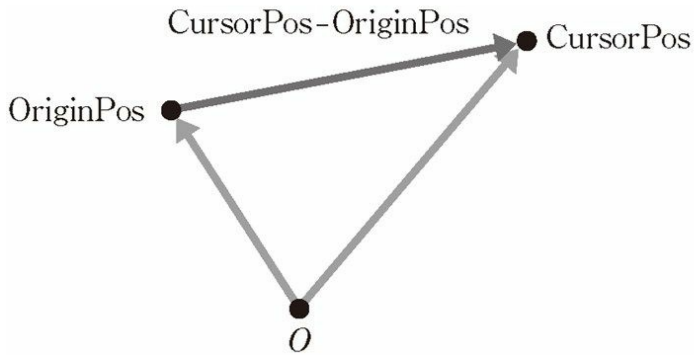


图 **4-1-6**　通过坐标间的减法得到坐标间的向量

得到了新的基向量 $\boldsymbol{i'}$ 之后，与 Ray_1_1.cpp 的处理相同，将向量 $\boldsymbol{i'}$（v2Forward）沿顺时针方向旋转 2 得到向量 $\boldsymbol{j'}$（v2Side）。下面就是通过向量 $\boldsymbol{i'}$、$\boldsymbol{j'}$ 计算图形的四个角的坐标，虽然此处也可以通过矩阵进行坐标变换，但是使用向量的加法运算会更加容易理解。首先，令图形左上角为 OriginPos 所在的位置，右上角为 CursorPos 所在的位置。那么左下角就等于 OriginPos 所在的位置加上向量 $\boldsymbol{j'}$（v2Side），右下角就等于 CursorPos 所在的位置加上向量 $\boldsymbol{j'}$（v2Side）。这是因为向量 $\boldsymbol{j'}$ 代表旋转后的图形的 y 坐标，而通过将左上角、右上角分别与向量 $\boldsymbol{j'}$ 相加，就可以求得图形下端的坐标（参考图 4-1-7）。

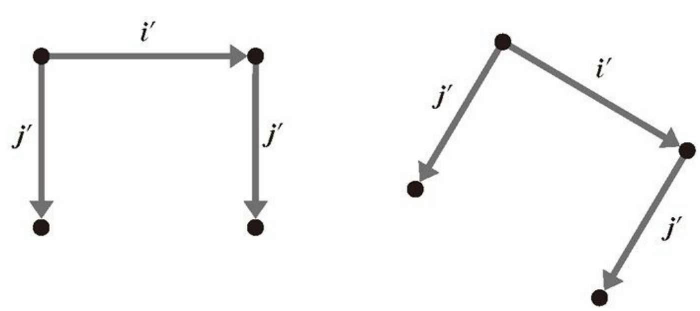


图 **4-1-7**　通过与向量 j' 相加得到下端的坐标

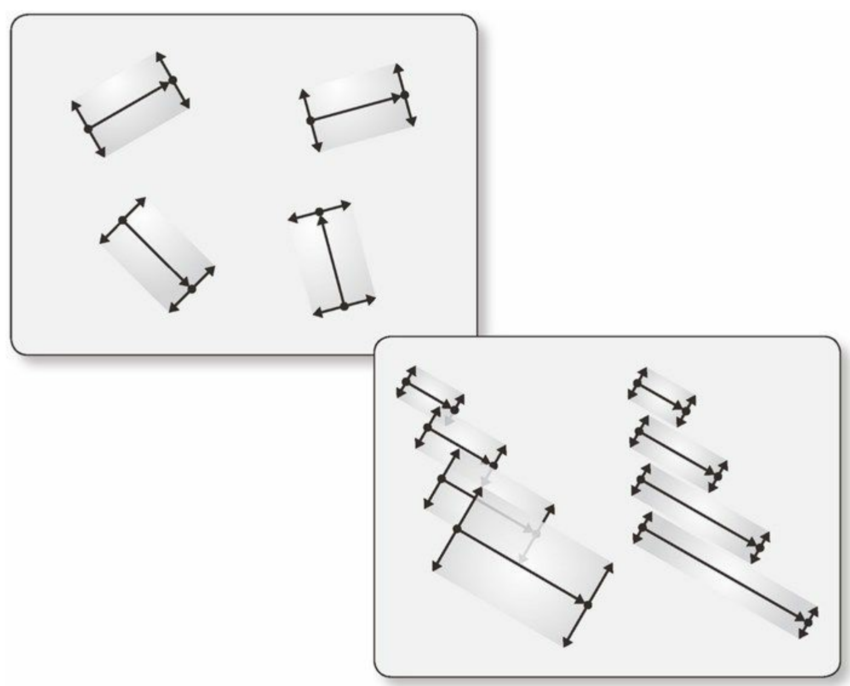
像这样，我们就实现了将物体沿某个向量旋

转，实现的过程中不仅没有使用正弦余弦三角函数，甚至连反三角函数都完全没有用到。这对减少程序的运算量是非常有好处的，特别是在 3D 场景中更加有优势，因此希望读者能牢记这些方法。

4.2　任意两点间的光线投射

Key Word　向量长度、单位向量





前一小节中我们学习了目标物体的位置固定时的旋转。本小节将讲解目标物体在任意位置时如何实现旋转。

作为不使用旋转矩阵或三角函数等的物体旋转的一个应用，本小节将尝试让光线在任意两点间投射。这一效果的实现是建立在 4.1 节的内容的基

础上的，因此建议读者先理解掌握上一小节的内容。

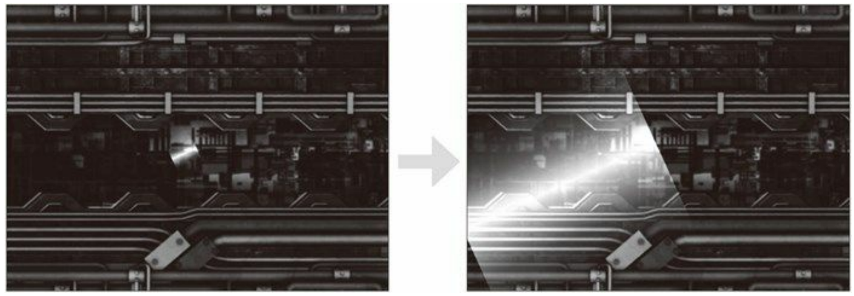


图 **4-2-1**　将不使用旋转矩阵或三角函数等的旋转应用于光线投射的程序

在真正开始执行光线投射程序之前，首先让我们来尝试实现一个光线的宽度可以根据长度联动变化的程序，如示例程序 Ray_2_1.cpp 所示，程序中的重点为 MoveRay 函数中的以下部分（代码清单 4-2-1）。

代码清单 **4-2-1**　光线宽度根据长度联动变化的 **MoveRay** 函数的主要部分（**Ray_2_1.cpp** 片段）

```
    v2Forward.x = ( float )( CursorPos.x - O
    v2Forward.y = ( float )( CursorPos.y - O
    v2Side.x = -v2Forward.y / 2.0f;
    v2Side.y =  v2Forward.x / 2.0f;

    v2Pos1.x = OriginPos.x - v2Side.x; v2Pos
    v2Pos2.x = CursorPos.x - v2Side.x; v2Pos
    v2Pos3.x = OriginPos.x + v2Side.x; v2Pos
    v2Pos4.x = CursorPos.x + v2Side.x; v2Pos
```

程序中首先将光线从起点指向终点的向量放入 v2Forward，也就是说 v2Forward 与光线的行进

方向是平行的。接下来，将 v2Forward 沿顺时针方向旋转 r 90 度），并将等于其长度一半的向量放入 v2Side，也就是说 v2Side 与光线的行进方向垂直。最后，使用 v2Side 向量，将图形的四个角通过以下方法计算出来（参考图 4-2-2）。

- 左上角 = 光线的起点 - v2Side
- 右上角 = 光线的终点 - v2Side
- 左下角 = 光线的起点 + v2Side
- 右下角 = 光线的终点 + v2Side

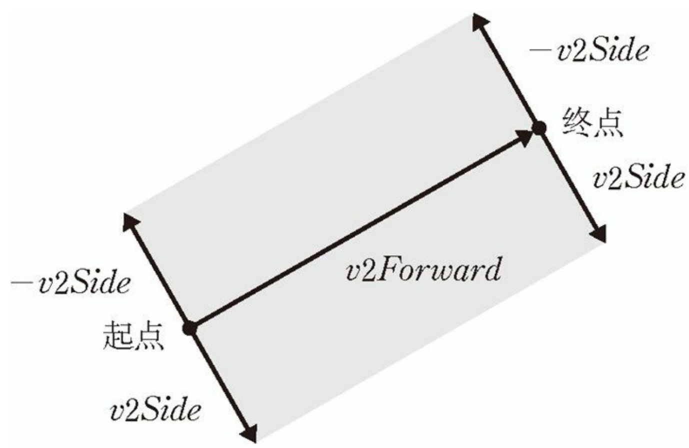


图 **4-2-2**　通过起点、终点及向量计算图形的四个角

由于 v2Side 代表的是长度为光线长度的一半，并且与光线行进方向垂直的向量，因此无论光线向任何方向投射，都可以通过向量的加减法，得到如图 4-2-2 所示的几点，最终绘制出一条宽度与长度联动变化的光线。请留意我们解决问题的核心思路是：要使光线可以向任意方向投射，就需要对光线图像进行旋转，而对光线图像进行旋转，只需准备一条与光线行进方向垂直的向量，并使用向量间的加减法（不需要三角函数或反三角函数）计算出图形的变化即可。

- 实现固定宽度的光线

接下来我们将介绍如何实现固定宽度的光线。示例程序 Ray_2_1.cpp 中所演示的这种光线，其宽度会随光线长度的增加而增加，这在现实中并不多见。现实中比较实用的是宽度固定的光线，如示例程序 Ray_2_1a.cpp 所示。

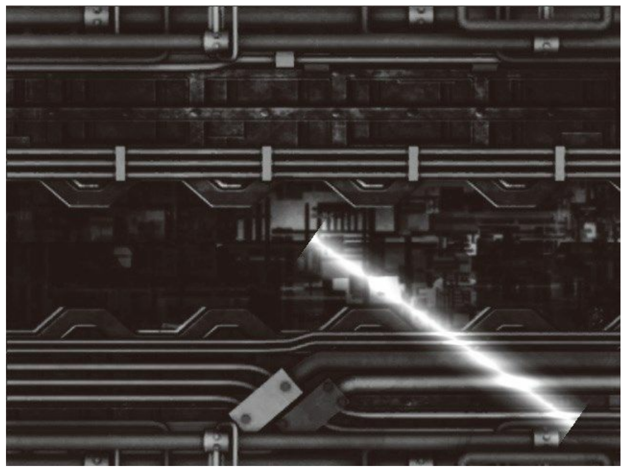


图 **4-2-3**　宽度固定的光线

该程序将原 Ray_2_1.cpp 中的

```
    v2Side.x = -v2Forward.y / 2.0f;
    v2Side.y =  v2Forward.x / 2.0f;
```

这两行进行了更改，如代码清单 4-2-2 所示。

代码清单 **4-2-2**　固定光线的宽度（**Ray_2_1a.cpp** 片段）

```
    fLength = sqrtf( v2Forward.x * v2Fo
```

```
v2Forward.x /= fLength; v2Forward.y
v2Side.x = -v2Forward.y * RAY_WIDTH
v2Side.y =  v2Forward.x * RAY_WIDTH
```

为了理解这部分处理，首先需要理解在上文的 Ray_2_1.cpp 中，光线宽度是如何跟随长度联动变化的。决定光线变化的要素有两个：与光线行进方向平行的向量 v2Forward 以及与光 线行进方向垂直的向量 v2Side。在 Ray_2_1.cpp 中，v2Forward 是光线的起点到终点的向量，因此 v2Forward 理所当然会随光线的长度变化而变化，而 v2Side 则是将 v2Forward 旋转 $\frac{\pi}{2}$ 并将长度变为一半后得到的，所以 v2Side 向量的长度也是动态变化的。另外，由于 v2Side 向量决定着光线的宽度（参考图 4-2-2），因此一旦 v2Side 向量的长度发生变化，光线的宽度也会随之变化。反过来想，为了让光线的宽度固定，其实只需要让 v2Side 向量的长度固定就行了。v2Side 向量的长度之所以会变化，是因为 v2Forward 向量的长度会发生变化，因此，为了固定 v2Side 的长度，我们有下面两个选择。

1. 只固定 v2Side 向量的长度，不固定 v2Forward 向量的长度

2. 将 v2Side 向量及 v2Forward 向量的长度都固定

示例程序 Ray_2_1a.cpp 中采用了方式 2，将两个向量的长度都固定了。这样正好可以对应与光线平行及垂直的一对向量。为此

Ray_2_1a.cpp 中将与光线平行的 v2Forward 向量作为了单位向量 。具体来说，就是根据勾股定理求得 v2Forward 向量的长度，然后将 v2Forward 的 *x* 分量、*y* 分量分别除以该长度。

```
    fLength = sqrtf( v2Forward.x * v2Fo
    v2Forward.x /= fLength;  v2Forward.
```

这样得到的 v2Forward 向量，就是与光线长度（起点到终点间的距离）无关的长度为 1 的向量。基于这样的 v2Forward 向量，并对 v2Side 向量做如下计算，最终就可以得到宽度为 RAY_WIDTH 的光线。

```
    v2Side.x = -v2Forward.y * RAY_WIDTH
    v2Side.y =  v2Forward.x * RAY_WIDTH
```

即代表将 v2Forward 向量按顺时针方向旋转 $\frac{\pi}{2}$，长为 RAY_WIDTH 的一半的向量。将 v2Side 向量分别加减光线的起点、终点坐标，就得到了宽度为 RAY_WIDTH 的光线（参考图 4-2-2）。

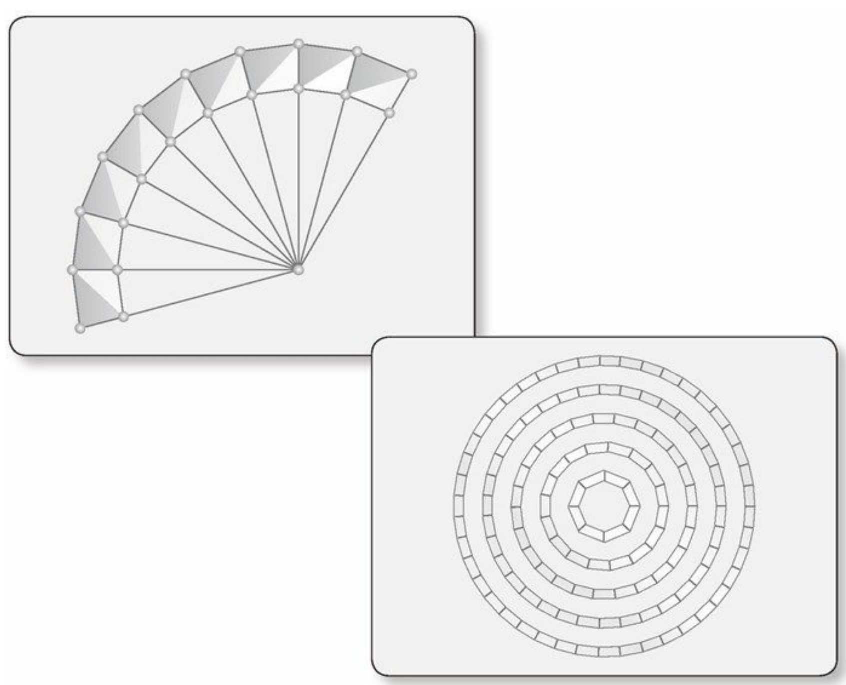
在本小节及前一小节中，为了实现简单的光线投射，虽然必须对光线的图形进行旋转、伸缩等处理，但我们尽量避免了使用反三角函数、旋转矩阵、放大缩小矩阵等。这种处理问题的方式，可以应用到游戏开发的很多

场景中，希望读者们可以牢记。

4.3　光线弯曲处理

Key Word　圆形、圆周长、伪影

RANK/normal



我们已经学习了如何表现直线形的光线，接下来就要考虑弯曲的光线的情况。在本小节，我们将学习如何绘制圆形的光线。

普通的光线都是笔直投射的，而本小节中我们将学习如何制作出弯曲的光线。



图 **4-3-1**　实现弯曲的光线的程序

例如，示例程序 Ray_3_1.cpp 就绘制出了圆形的光线。该圆形光线以画面中央为中心，而且光线会始终穿过鼠标指针所在的位置。程序中重要的部分是 MoveRay 函数中的以下代码（代码清单 4-3-1）。

代码清单 **4-3-1**　绘制圆形光线的 **MoveRay** 函数的主要部分（**Ray_3_1.cpp** 片段）

```
    nDivideNum = ( int )( ( 2.0f * PI * r )
    if ( nDivideNum > MAX_DIVIDE_NUM ) nDivi
    fAngleDiff = 2.0f * PI / nDivideNum;
    r1 = r - ( RAY_WIDTH / 2.0f );
    r2 = r + ( RAY_WIDTH / 2.0f );

    fAngle1 = 0.0f;
    fAngle2 = fAngleDiff;
    for ( i = 0; i < nDivideNum * 4; i += 4
        v2Pos[i    ].x = r2 * cosf( fAngle1
        v2Pos[i    ].y = r2 * sinf( fAngle1
        v2Pos[i + 1].x = r2 * cosf( fAngle2
        v2Pos[i + 1].y = r2 * sinf( fAngle2
        v2Pos[i + 2].x = r1 * cosf( fAngle1
        v2Pos[i + 2].y = r1 * sinf( fAngle1
        v2Pos[i + 3].x = r1 * cosf( fAngle2
        v2Pos[i + 3].y = r1 * sinf( fAngle2
```

```
            fAngle1 += fAngleDiff;
            fAngle2 += fAngleDiff;
        }
```

程序中首先确定了如何将由光线构成的圆形（或者可以说圆环）按角度方向分割，并将分割数放入了变量 nDivideNum 中。因为计算机一般都不支持直接绘制曲线，只能自由绘制三角多边形，因此将光线的圆环按角度方向分割为若干个很小的三角多边形，就可以通过三角多边形最终绘制出圆环（参考图 4-3-2）。

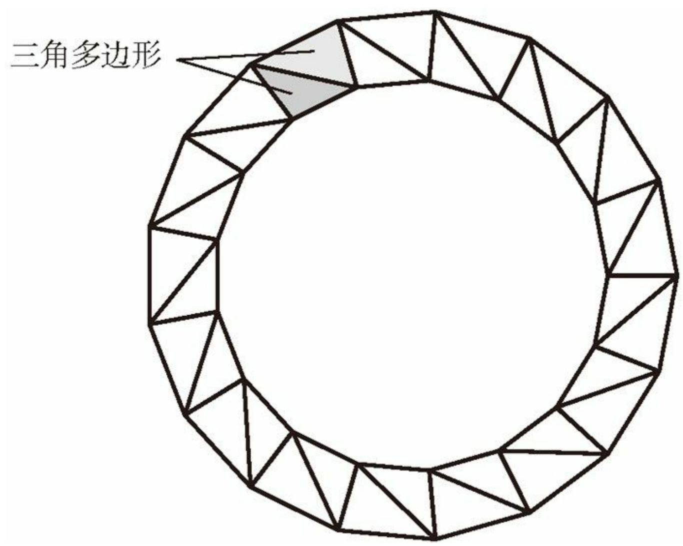


图 **4-3-2**　通过三角多边形绘制圆环

决定分割数的方法有很多，这里按照分割后圆周长的一份约为 10 像素来决定分割数。由于半径为 r 的圆的圆周长为 $2\pi r$，按照一份圆周长为 10 像素来计算，圆环会被分割为 $\frac{2\pi r}{10}$ 份，也就是上面程序中的

```
    nDivideNum = ( int )( ( 2.0f * PI * r )
```

这一行。但是如果这个分割数过大，所需要的顶点数也会增多，这时就可能有缓冲区溢出的风险，因此这里对分割数设置了 MAX_DIVIDE_NUM 这一上限。

```
    if ( nDivideNum > MAX_DIVIDE_NUM ) nDivi
```

这样分割数就不会超过上限值 MAX_DIVIDE_NUM 了。

然后根据已经决定的分割数，计算出每一份多边形的角度，并将其放入变量 fAngleDiff 中。这个角度通过圆一周的角度 2π 除以分割数就可以得到。

```
    fAngleDiff = 2.0f * PI / nDivideNum;
```

接下来设置了 r1 与 r2 两个半径。其中 r1 是光环内圆的半径，r2 是光环外圆的半径。然后考虑通过这两个半径以及刚才计算得到的分割数，来用四角形表现分割出的每份图形。实际上程序中绘制每份图形的是梯形（参考图 4-3-3），这样就可以将一个圆形转换为若干个多边形进行绘制了。而只要将圆分割成足够小的多边形，肉眼看

上去就不会有问题。

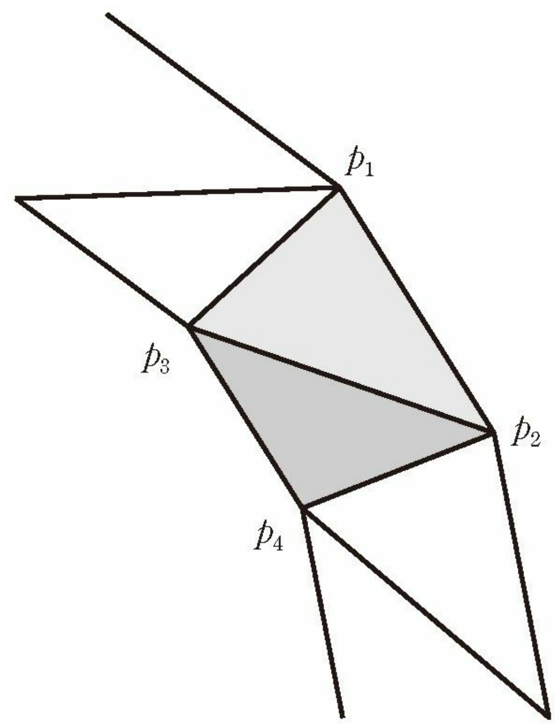


图 **4-3-3**　将圆环分割后得到的图形为梯形

然后，分割数为多少，就循环多少次。用梯形表示角度在 fAngle1 与 fAngle2 之间、半径在 r1 与 r2 之间的区域，并计算每个梯形的四个角的坐标。此时假设

- p_1 是半径 r2 且角度为 fAngle1 的位置
- p_2 是半径 r2 且角度为 fAngle2 的位置
- p_3 是半径 r1 且角度为 fAngle1 的位置
- p_4 是半径 r1 且角度为 fAngle2 的位置

这样就可以通过 p_1、p_2、p_3、p_4 表示四个角的坐标（参考图 4-3-3）。例如由于 p_1 到圆心的距离为 r2，且角度为 fAngle1，因此令 r2=r_2、

fAngle1 =θ_1、圆心坐标 CenterPos=(c_x , c_y)，根据三角函数的定义，有

$$p_{1x} = r_2 \cos\theta_1 + c_x$$
$$p_{1y} = r_2 \sin\theta_1 + c_y$$

这对应代码清单 4-3-1 中的

```
        v2Pos[i    ].x = r2 * cosf( fAngle1
        v2Pos[i    ].y = r2 * sinf( fAngle1
```

这一部分，p_2、p_3、p_4 与此同理。

上文介绍了绘制圆形光线的原理，但是这个方法还存在一点问题。在示例程序 Ray_3_1.cpp 中，将激光图片用于纹理贴图（texture mapping）时，仅 *v* 坐标会被设置为有效值，而 *u* 坐标则始终为 0（具体请参考 Ray_3_1.cpp 内部，DrawQuadranglePic 函数内）。这是由于如果对 *u* 坐标也设置一个有效值的话，当圆周变小时，梯形会变得细长，此时由梯形对角线切分而成的三角多边形也会变得极度细长，那么仅根据 3 个顶点的 *uv* 坐标将无法正确计算三角多边形内部的 *uv* 坐标，就会发生所谓的“伪影”（artifacts）现象。

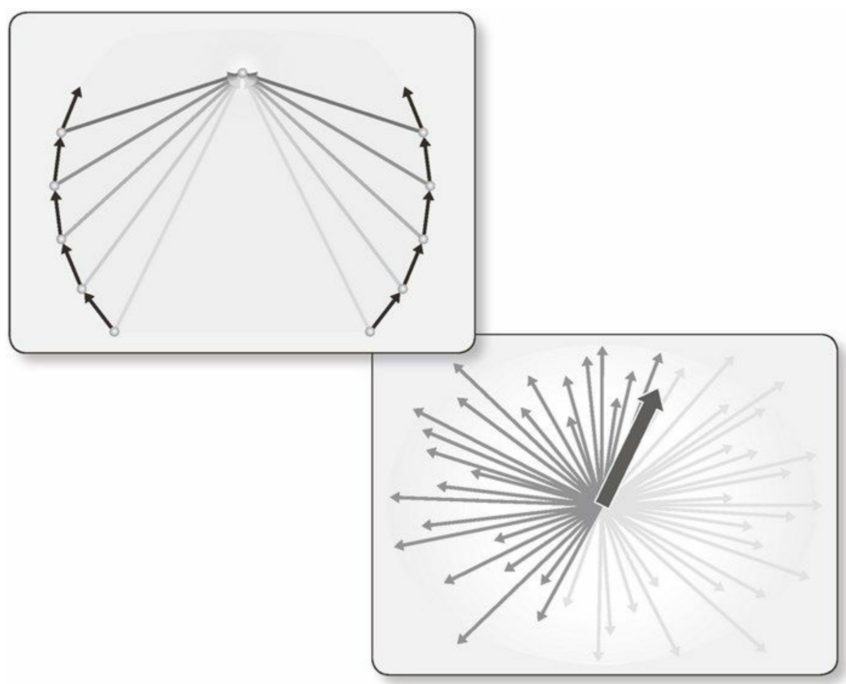
虽然从数学上来看是分割成了等长距离（如 10 像素）的圆周，但其实分割得到的梯形的长度是会变化的，因为分割数始终为整数，特别是圆周很小时分割数也会很小，此时如果分割数产生变化，圆周长度的变化就会非常清晰地反映到梯形长度的变化上。但是在纹理贴图时，如果 *u* 坐标始终固定为 0，就没有办法作出螺旋状的激光。因此，如果想实现螺旋状的圆形激光，还需要对

多边形进行更加精细的分割。

4.4 实现带追踪效果的激光

Key Word 左右判定、外积、旋转速度

RANK/hard



如果要选出射击游戏中最具代表性的激光武器，那么一定少不了带追踪效果的激光。本小节就来一起实现可以对鼠标指针进行追踪的激光吧。

本小节将讲解如何实现追踪目标物体并可弯曲的激光（追踪激光）。这种激光在很多射击游戏及动作游戏中都会经常出现。



图 **4-4-1**　追踪目标物体并可弯曲的激光

姑且不论可弯曲的激光在现实世界中究竟有没有可能实现，在程序的虚拟世界中，通过简单的算法就可以制作出来，请参考示例程序Ray_4_1.cpp。运行这个程序后，原本从画面左端笔直发射的激光会追踪鼠标指针并弯曲，看起来有点像蚯蚓在笨拙地爬行。之所以会显得笨拙，主要还是因为算法进行了一定的简化，这在下文中会详细说明。为了理解 Ray_4_1.cpp 的原理，我们首先还是先看程序的代码吧。

Ray_4_1.cpp 的 MoveRay 函数中，会将激光按某个长度（程序中是 SEGMENT_LEN）分割为很多小节，然后对每节激光设置不同的方向。所产生的效果就是刚开始激光向水平方向发射，当鼠标指针在激光行进方向左侧时，激光就向左转向（图 4-4-2 左），而当鼠标指针在右方时就向右转向（图 4-4-2 右），激光就好像在追踪鼠标指针一样。由于每次转向的旋转速度是固定的（程序中是 ANGLE_SPEED），因此鼠标指针如果位于偏离激光行进方向很远的位置，就可以避开激光的追踪。

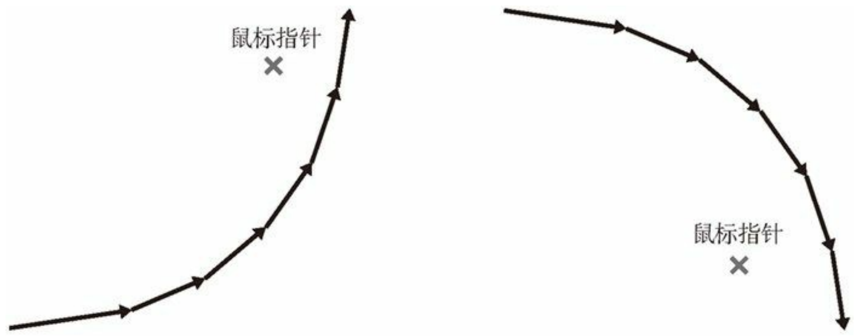


图 **4-4-2**　将激光按某个长度分割为多个小节并更改其方向实现弯曲

为了实现这个效果，从数学角度来看最重要的是判断目标物体是在激光行进方向的左侧还是右侧。我们可以将其转化为“一个点在一个有向线段的左侧还是右侧”这样一个数学问题（参考图4-4-3）。

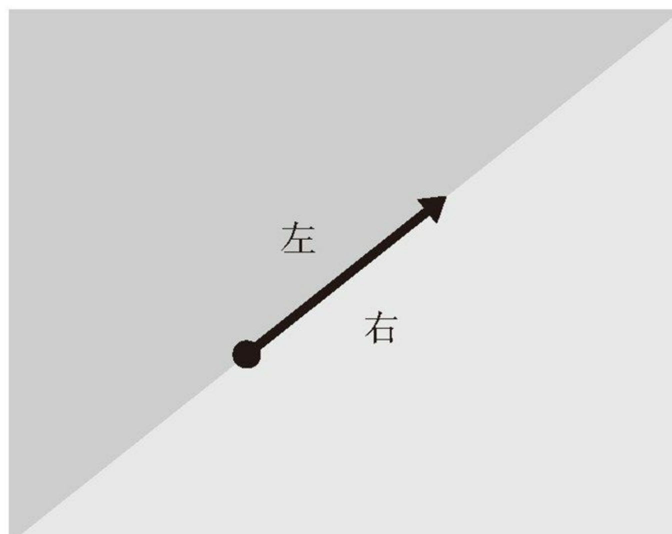


图 **4-4-3**　一个点在线段的左侧还是右侧

可能数学好的读者会想到，只要将有向线段朝向的角度，以及从当前的激光位置所看到的目标物的角度进行比较就可以判断出左右了。但是这并不是一个好方法，首先一旦涉及角度，就必须使用计算量较大的反三角函数，会导致速度变慢。其次，上面所说的角度，一般都是相对 x 轴的角

度，将这样的两个角度进行比较时，判断一方在另一方的左侧还是右侧，还是比较麻烦的。至少仅对比角度数值的大小，是无法直接判断是在左侧还是右侧的。因此 Ray_4_1.cpp 的 MoveRay 函数中就使用了其他方法来计算角度。具体来说就是执行了下面的判定。

代码清单 **4-4-1**　左右侧判定（**Ray_4_1.cpp** 片段）

```
073  |        if ( ( v2Forward.x * v2Aim.y - v2For
```

v2Forward 是代表激光行进方向的向量，v2Aim 是从当前激光位置看到的指向鼠标指针的向量（参考图 4-4-4）。

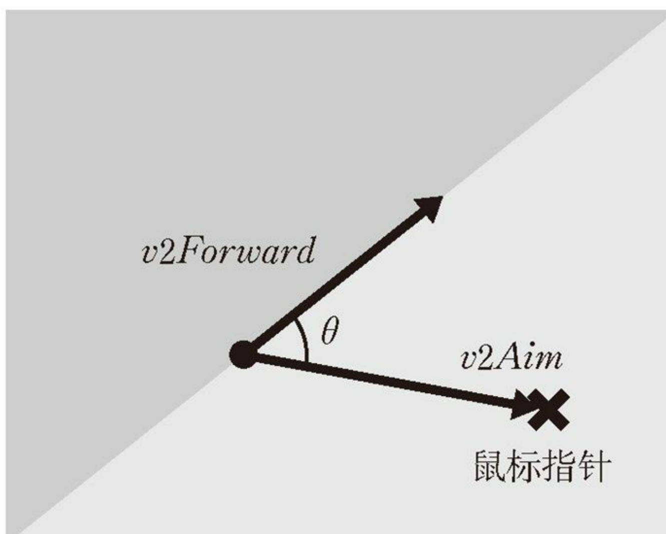


图 **4-4-4**　激光行进方向向量及指向鼠标指针的向量

令 v2Forward=(F_x , F_y)、v2Aim=(A_x , A_y)，代码清单 4-4-1 的判断语句其实就是在检查 $F_x A_x$ -F_y

A_y 所得到的值的符号。这是为什么呢？可能很多人会首先想到向量的外积（的一部分），而从这个角度来考虑也是最容易理解的。所谓向量的外积，比如有两个三元向量 $a=(a_x, a_y, a_z)$ 与 $b=(b_x, b_y, b_z)$，则有

$$a \times b = (a_y b_z - a_z b_y,\ \ a_z b_x - a_x b_z,\ \ a_x b_y - a_y b_x)$$
$$|\boldsymbol{a}||\boldsymbol{b}|\sin\theta \cdot \hat{\boldsymbol{n}}$$

所得到的结果是一个向量，θ 是向量 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 的夹角，$\hat{\boldsymbol{n}}$ 的长为 1，并且同时与向量 $\boldsymbol{a}$ 、 $\boldsymbol{b}$ 都垂直（这样的向量会有两条，具体为指向哪个方向的向量则需要根据坐标系来看）。

虽然我们要实现的追踪效果只需要考虑 2D 平面的情况，但为了计算外积，这里将其暂时扩展到 3D，令 v2Forward=(F_x, F_y, 0)、v2Aim=(A_x, A_y, 0)。取这两个向量的外积，可以看到外积的 x 分量（向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 时为 $a_y b_z - a_z b_y$）及 y 分量（向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 时为 $a_z b_x - a_x b_z$）中，都有向量 z 分量的乘法运算（两个向量的 z 分量都为 0），因此外积的 x 分量与 y 分量都为 0。只有外积的 z 分量（向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 时为 $a_x b_y - a_y b_x$）在一般情况下不为 0，等于 $F_x A_x - F_y A_y$。这就是之前程序中所写的判断语句。

根据上面的等式可知，外积的结果等于 $|\boldsymbol{a}||\boldsymbol{b}|\sin\theta \cdot \hat{\boldsymbol{n}}$。如果将 $\hat{\boldsymbol{n}}$ 作为固定指向某个特定方向的单位向量的话，由于 $|\boldsymbol{a}|$ 与 $|\boldsymbol{b}|$，即两向量的长度不可能为负，所以 $F_x A_y - F_y A_x$ 的值的符号就由两向量的夹角 θ 的 $\sin\theta$ 的符号决定。如果 $F_x A_y - F_y A_x$ 的值的符号反转，则 $\sin\theta$ 的符号也反转。由于 $\sin\theta$ 是奇函数，$\sin\theta$ 的符号与 θ 的符号相同（只限 $-\pi < \theta < \pi$ 时），因此最终 $F_x A_y - F_y A_x$ 的符号就决定了两向量的夹角 θ 的符号，

也就是说，只要看 $F_x A_y - F_y A_x$ 的结果，就能对向量做左右判定。上面的说明可能有点长，希望大家可以正确理解。

既然已经明白了为什么 $F_x A_y - F_y A_x$ 的符号可以作为左右判定的条件，接下来就可以在程序中使用这个值。那么当该值为正时，究竟会判定为左侧还是右侧呢？由于这个问题是放在之前所构想的 2D 场景中来考虑的，用外积可能不太好理解。因此我们可以先考虑 $F_x = 1$、$F_y = 0$ 这一特殊场景。此时激光会笔直向右方行进，如果目标方向的 y 分量 A_y 为正，则在行进方向右侧；反之，如果 A_y 为负，则在行进方向左侧。同时当 $F_x = 1$，$F_y = 0$ 时，$F_x A_y - F_y A_x$ 的值就等同于 A_y 的值，即 $F_x A_y - F_y A_x$ 的值为正时在行进方向右侧，为负则在行进方向左侧。因此在 Ray_4_1.cpp 的 MoveRay 函数中，如果 $F_x A_y - F_y A_x$ 的值为正，就认为鼠标指针在激光行进方向右侧，随着角度值的增加，行进方向就会沿顺时针转向（右转）。相反当 $F_x A_y - F_y A_x$ 的值为负时，则认为鼠标指针在激光行进方向左侧，随着角度值的减少，行进方向就会沿逆时针转向（左转）。通过这种方式，最终就实现了无论鼠标指针在行进方向的任意一侧，激光都会对鼠标指针进行追踪的效果。

代码清单 **4-4-2　MoveRay** 函数（**Ray_4_1.cpp** 片段）

```
        if ( ( v2Forward.x * v2Aim.y - v2For
            fAngle += ANGLE_SPEED;
        }
        else {
            fAngle -= ANGLE_SPEED;
        }
```

像上面这样，根据追踪对象在行进方向的左侧还是右侧，来以一定旋转速度转向的算法，其实还无法做出很流畅的追踪效果。Ray_4_1.cpp 中的激光，之所以看起来像蚯蚓在扭来扭去，其最大原因就是无论追踪对象偏离行进方向很远，还是在行进方向正前方，都使用了相同的旋转速度转向。特别是当追踪对象大致位于行进方向正前方时，如果仍使用很高的旋转速度转向，就会让激光在一次转向过程中一会儿向左转一会儿向右转，产生大量无用的扭动，所呈现的视觉效果也很差（参考图 4-4-5）。

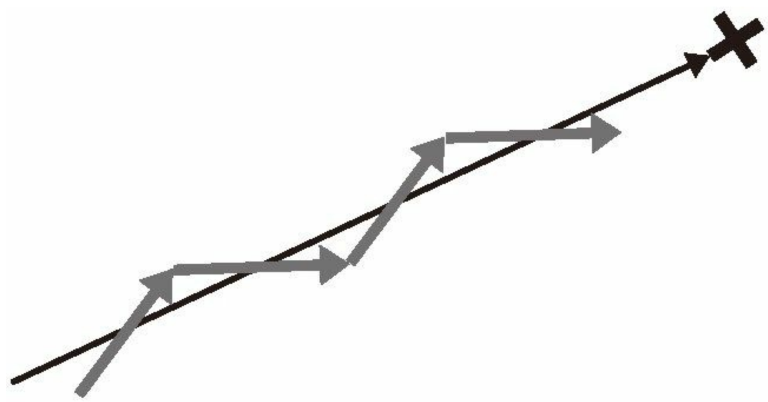


图 **4-4-5**　如果不考虑行进方向与追踪对象的位置关系，就会造成不自然的扭动

针对追踪对象与行进方向的位置关系所产生的问题，示例程序 Ray_4_1a.cpp 对原程序做了修正，让旋转速度可以动态变化（当物体在行进方向正前方时几乎不转向）。程序中的重点是 MoveRay 函数的以下部分（代码清单 4-4-3）。

代码清单 **4-4-3**　旋转速度可动态变化的 **MoveRay** 函数（**Ray_4_1a.cpp** 片段）

```
        fLength = sqrtf( v2Aim.x * v2Aim.x +
```

```
        v2Aim.x /= fLength;  v2Aim.y /= fLer
        fCross = v2Forward.x * v2Aim.y - v2F
        fAngle += ANGLE_SPEED * fCross / SEG
```

可以看到 Ray_4_1.cpp 中的 if 语句没有了，而且还向激光投射角度 fAngle 增加了一个固定值。那么所增加的值有什么意义呢？

还是按照顺序来看吧。首先代码清单 4-4-3 中最开始的

```
        fLength = sqrtf( v2Aim.x * v2Aim.x +
        v2Aim.x /= fLength;  v2Aim.y /= fLer
```

这两行程序将以激光为起点、指向追踪目标鼠标指针的向量（v2Aim），变换为了长为 1 的向量，即单位向量。在此基础上通过

```
        fCross = v2Forward.x * v2Aim.y - v2F
```

计算出了 v2Forward 与 v2Aim 的外积的 z 分量。我们在前文中已经讲过，取向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 的外积时，其外积结果的长度为 $|\boldsymbol{a}||\boldsymbol{b}|\sin\theta$ 。刚才已经将 v2Aim 变换为了单位向量，那么此时变量 fCross 的值，就是将向量 v2Forward 的长，乘以 v2Forward 与 v2Aim 的夹角 θ 的 $\sin\theta$ 所得到

的。让我们再看看 Ray_4_1a.cpp 中更往上的部分。

```
    v2Forward.x = SEGMENT_LEN * cosf( fAngle )
    v2Forward.y = SEGMENT_LEN * sinf( fAngle )
```

可以看到 v2Forward 向量的长被设置为了 SEGMENT_LEN。因此

```
        fAngle += ANGLE_SPEED * fCross / SEG
```

这一行中的 fCross / SEGMENT_LEN 就等同于 $\sin\theta$ 。前面已经提到过，$\sin\theta$ 是奇函数， 在 $-\pi < \theta < \pi$ 的范围内 θ 的符号就是 $\sin\theta$ 的符号，θ 为 0 则 $\sin\theta$ 也为 0，当追踪对象在行进方向的正对面时，$\theta=0$，$\sin\theta=0$；而对象偏离行进方向越远，$\sin\theta$ 也就越大（如果是负方向则越小）。上面的程序中将 $\sin\theta$ 乘以常数 ANGLE_SPEED 所得的值作为了旋转速度，这样就可以根据追踪对象偏离行进方向的具体情况来控制旋转速度了。

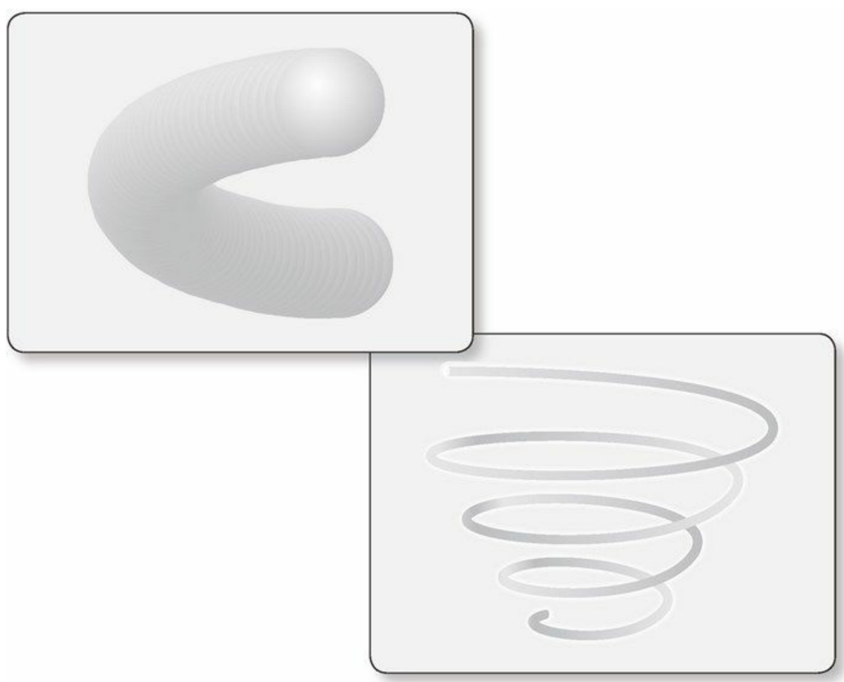
但是上例中旋转速度的最大值是 ANGLE_SPEED，当 $\sin\theta = \pm 1$，即 θ 为 $\pm\frac{\pi}{2}$ 时就产生了最大旋转速度。如果 θ 的绝对值超过了 $\frac{\pi}{2}$，根据上例可知，角度（绝对值）越大，旋转速度越小，当 θ 的绝对值为 π 时旋转速度完全减小至 0。这在实际程序中，就相当于追踪对象已经移动到了行进方向的后面。而角度越接近 π，追

踪对象就越接近正后方。当追踪对象移动到正后方时，追踪效果完全停止，这样的处理也是符合常理的。

4.5 [进阶] 绘制大幅度弯曲的曲线时的难点

Key Word 曲率、曲线的粗细、插值曲线、反射

RANK/very hard



在绘制曲线时，必须要考虑曲线的粗细问题。本小节就来一起探究这一问题的难点。

计算机在绘制弯曲的光线等有宽度的曲线时，会遇到一些问题，本小节将介绍如何解决这些问题。例如在 4.4 节中，在曲线的行进方向建立垂

直向量（也就是法线向量），然后将其加到曲线的位置向量，通过这种方法实现了有宽度的曲线的绘制（参考图 4-2-2）。这个方法在曲线的弯曲程度（即曲率）不是很大的时候没有什么问题，但是当曲率增大到一定程度后，多边形会因为严重扭曲而大量重叠，如果光线是半透明绘制的话，光线的一部分就会变得特别亮（参考图 4-5-1）。

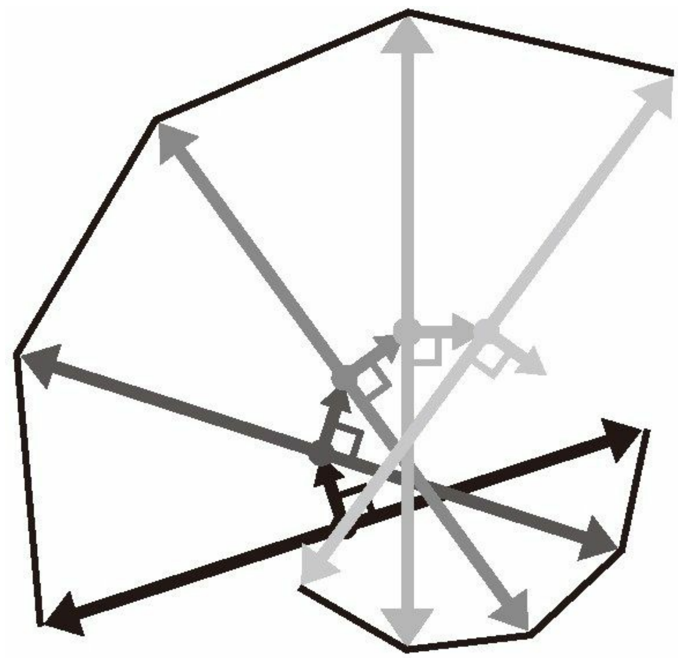


图 **4-5-1**　多边形产生重叠

解决这一问题有多种方法，其中一个方法是检测多边形重叠的部分，对重叠部分不进行绘制。在曲线的曲率不是很大时，采用这种方法能获得不错的显示效果。

但是如果曲率进一步增加，与曲线弯曲的一侧相对的另一侧（即曲线的外侧）会变得细长。这是因为将曲线沿行进方向等距离分割时，曲线外侧的长度会变长，其实这个问题本质上还是因为分割数不够而引起的（参考图 4-5-2）。

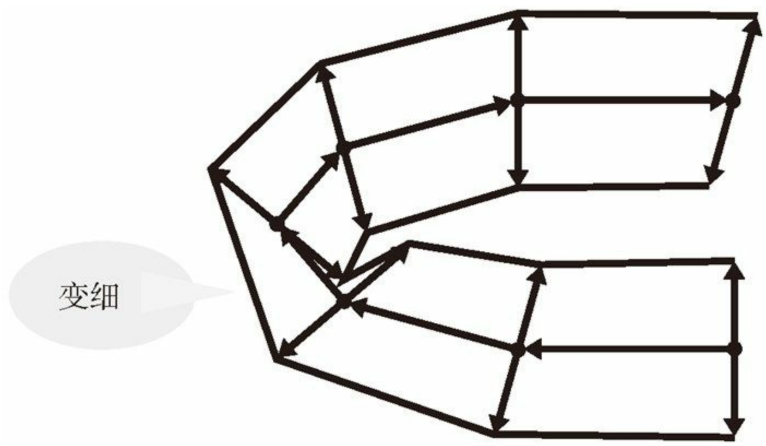


图 **4-5-2**　曲率变大时外侧变得细长

解决这一问题也有不少方法，其中一个方法是让曲线按照行进方向分割的宽度，根据曲率的不同进行变化。如果曲率小，那么分割数也少；而如果曲率大，则会进行更加细密的分割。但是这种情况下，沿用 4.4 节介绍的以一定速度旋转曲线的处理方式来绘制曲线会变得非常困难。因为需要首先将运动轨迹表示为矩阵序列，然后作出一条矩阵序列所通过的插值曲线（如样条曲线 spline curve 等），并使用可变齿距将差值曲线分割等一系列操作，非常麻烦。

不仅如此，如果考虑到反射等可以让曲线以接近 180 度角折返的情况，从数学角度上来讲就等于允许一个不连续的微分系数，情况将变得更加复杂。此时即便将曲线沿法线向量方向扩宽，也依然会造成反射部分中曲线的中心裸露出来，从而形成一个难看的缺口（参考图 4-5-3）。

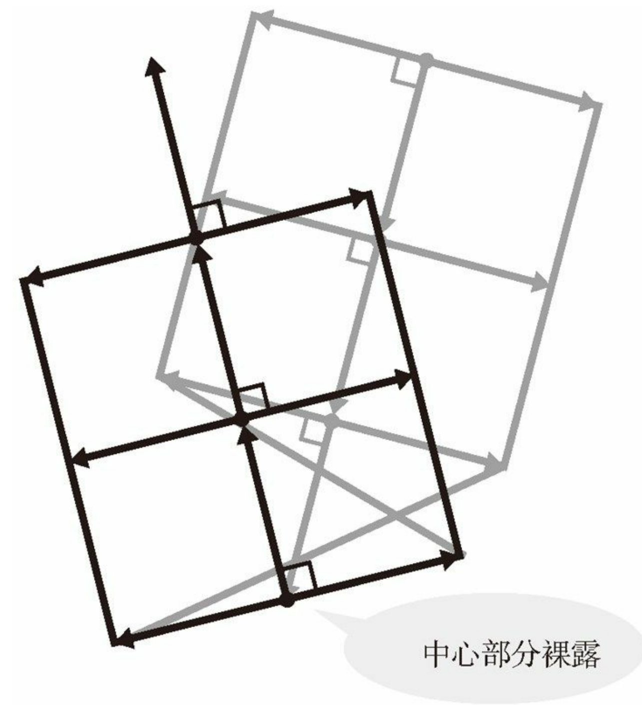


图 **4-5-3**　反射部分中曲线的中心部分裸露

如果想要绘制光线，这显然不是我们想要的效果。我们希望即使以接近 180 度的角反射时，也能绘制出曲线中心部分两侧都有图像的正常光线。这个问题当然也有解决方法，比如我们可以将曲线分割后的接缝处用圆形填充。这样处理后，无论曲线以何种角度弯曲，都可以保证曲线的宽度不会小于圆形的直径（参考图 4-5-4）。

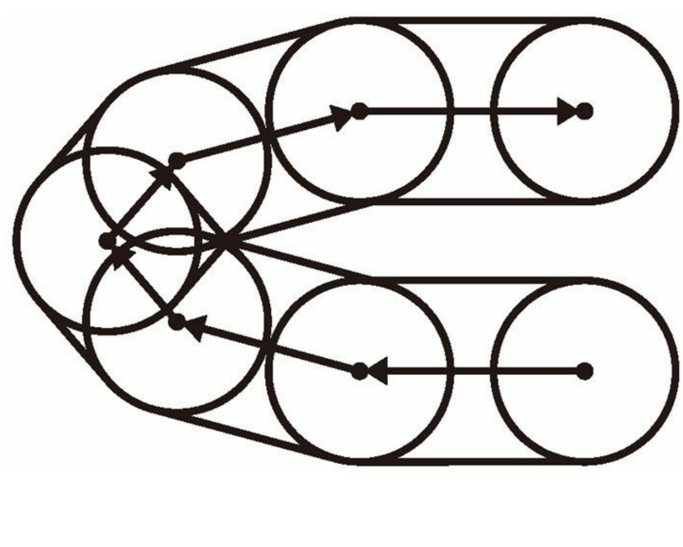


图 **4-5-4**　接缝处用圆形填充以保证曲线的宽度

虽然通过这种方法可以方便地实现曲线以任意角度弯曲，不过仍然存在缺点。比如在绘制光线时，由于在接缝处填充了圆形，圆形之间必定会有重叠的部分，因此如果要绘制亮度均匀的曲线就比较难。

综上可以看出，曲线随着其曲率的增大，绘制的难度也变得越来越大。绘制不透明的曲线还好说，如果想要绘制光线等有半透明效果的曲线，以之前例子中普通的图片素材填充是不可能实现的，而要根据亮度计算的结果准备图片素材，比如制作第一条曲线时只要根据需求准备单独的图片即可，而制作第二条曲线所需的图片就需要根据上一条曲线所计算得到的透明度重新准备。在绘制曲线时类似这样的小技巧还有不少，总之要制作完美的曲线效果是非常繁琐的。上面所讲的各种绘制曲线的方法中，并不存在一种理想的解决方案，还是需要根据具体需求进行选择，尤其是需要考量一下游戏中是不是真的需要这么精细的效果，必要的话可以适当做一些妥协。

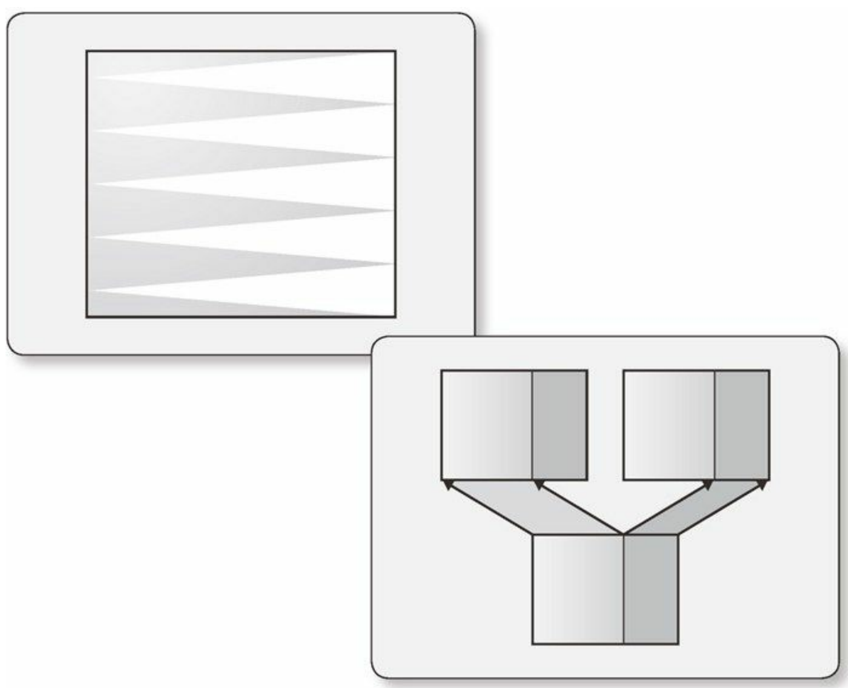
如果拓展到 3D 空间，那么绘制一条有宽度的曲线就会有更多问题出现。根据视角的不同，3D 曲线会有多种多样的变化，因此绘制一条粗细固定的 3D 曲线，要比 2D 复杂很多。特别是当曲线完全与视线平行，即曲线以视点为起点一直向内延伸时，即使将曲线按照一定间距分割，最终呈现的效果也还是像在同一坐标系内，这时即使想将各点的坐标落在同一 2D 平面上来计算其法线向量也会存在一些问题，使曲线具有宽度本身都会变得非常困难。如果不注意上面的问题就去绘制曲线，结果就会让曲线看起来好像一条轻飘飘的绸带。不过这个问题是有一些解决方法的，但是超出了本书的研究范围，这里就不做介绍了。

第 5 章　画面切换效果

5.1　水平扫描式画面切换

Key Word　三角多边形、纹理素材、**uv** 坐标

RANK/normal



文字冒险及 **RPG** 游戏中进行场景转换时，经常会使用将两张图像进行切换的方式。在本小节，就让我们来一起学习最基本的水平方向的画面切换吧。

本小节将讲解画面切换效果中最基本的水平方向扫描式画面切换。



图 **5-1-1**　水平方向扫描式画面切换程序

示例程序 Wipe_1_1.cpp 实现了这一效果，程序中的重点是以下部分。

代码清单 **5-1-1**　进行水平方向画面切换的主要部分（**Wipe_1_1.cpp** 片段）

```
int InitChangingPictures( void )
{
    fBoundary_x = 0.0f;
    fBoundary_vx = 10.0f;

    return 0;
}


int DrawChangingPictures( void )
{
    fBoundary_x += fBoundary_vx;
    if ( fBoundary_x < 0.0f ) {
        fBoundary_x = 0.0f;
        fBoundary_vx = -fBoundary_vx;
    }
    if ( fBoundary_x > VIEW_WIDTH ) {
        fBoundary_x = VIEW_WIDTH;
        fBoundary_vx = -fBoundary_vx;
    }
```

```

    Draw2DPolygon( 0.0f,                0.0f,
                   0.0f,         VIEW_HEIGHT,
                   fBoundary_x,         0.0f,
                   &g_tPic2 );
    Draw2DPolygon( fBoundary_x,         0.0f,
                   0.0f,         VIEW_HEIGHT,
                   fBoundary_x, VIEW_HEIGHT,
                   &g_tPic2 );
    Draw2DPolygon( fBoundary_x,         0.0f,
                   fBoundary_x, VIEW_HEIGHT,
                   VIEW_WIDTH,          0.0f,
                   &g_tPic1 );
    Draw2DPolygon( VIEW_WIDTH,          0.0f,
                   fBoundary_x, VIEW_HEIGHT,
                   VIEW_WIDTH,  VIEW_HEIGHT,
                   &g_tPic1 );

    return 0;
}
```

程序首先在 InitChangingPictures 函数中对两张图片的边界线位置 fBoundary_x 及边界线的移动速度 fBoundary_vx 进行了初始化。fBoundary_x 被设置为

```
    fBoundary_x = 0.0f;
```

图片边界线的初始位置在画面左端。而 fBoundary_vx 为

```
    fBoundary_vx = 10.0f;
```

图片边界线会以每帧 10 像素的速度向画面右方移动。

在接下来的 DrawChangingPictures 函数中，会实际移动边界线。其中用到了 1.1 节中物体接触到画面两端就折返的方法，让边界线在画面两端之间循环往复运动。关于这一部分的详细处理，请参考 1.1 节。

然后就是最关键的部分，即实际使用 2D 多边形渲染两张图片（56～71 行）。虽然多边形本身的形状并没有什么统一的规范，不过大部分显卡只支持渲染三角多边形（形状为三角形的多边形），本例也是通过组合三角多边形来渲染了两张图片。由于两张图片最终要渲染的区域都是长方形，渲染一张长方形图片需要两个三角多边形，因此渲染两张图片共需要 2×2=4 个三角多边形。在程序中对应以下部分。

```
056 |     Draw2DPolygon( 0.0f,                0.0f,
057 |                    0.0f,         VIEW_HEIGHT,
058 |                    fBoundary_x,         0.0f,
059 |                    &g_tPic2 );
060 |     Draw2DPolygon( fBoundary_x,         0.0f,
061 |                    0.0f,         VIEW_HEIGHT,
062 |                    fBoundary_x, VIEW_HEIGHT,
063 |                    &g_tPic2 );
064 |     Draw2DPolygon( fBoundary_x,         0.0f,
065 |                    fBoundary_x, VIEW_HEIGHT,
066 |                    VIEW_WIDTH,          0.0f,
067 |                    &g_tPic1 );
068 |     Draw2DPolygon( VIEW_WIDTH,          0.0f,
069 |                    fBoundary_x, VIEW_HEIGHT,
070 |                    VIEW_WIDTH,  VIEW_HEIGHT,
071 |                    &g_tPic1 );
```

其中 Draw2DPolygon 函数如下所示。

代码清单 **5-1-2　Draw2DPolygon** 函数（**Wipe_1_1.cpp** 片段）

```
int Draw2DPolygon( float x1, float y1, float
                   float x2, float y2, float
                   float x3, float y3, float
                   TEX_PICTURE *pTexPic );
```

各参数的意思如下。

- x1, y1 : 三角多边形的顶点 1 的 x、y 坐标
- u1, v1 : 三角多边形的顶点 1 的 u、v 坐标
- x2, y2 : 三角多边形的顶点 2 的 x、y 坐标
- u2, v2 : 三角多边形的顶点 2 的 u、v 坐标
- x3, y3 : 三角多边形的顶点 3 的 x、y 坐标
- u3, v3 : 三角多边形的顶点 3 的 u、v 坐标
- pTexPic : 图片素材的指针

- 能有效利用多边形的 ***uv*** 坐标

考虑到有很多读者对多边形贴图还不是很了解，这里对 ***uv*** 坐标 进行一些简单的介绍。多边形在 3D 渲染时经常使用，能够将纹理图片（texture）贴到上面。因此多边形的各顶点都会与纹理中的某个坐标相对应。而各顶点在图片中的坐标就称为 *uv* 坐标（图 5-1-2）。

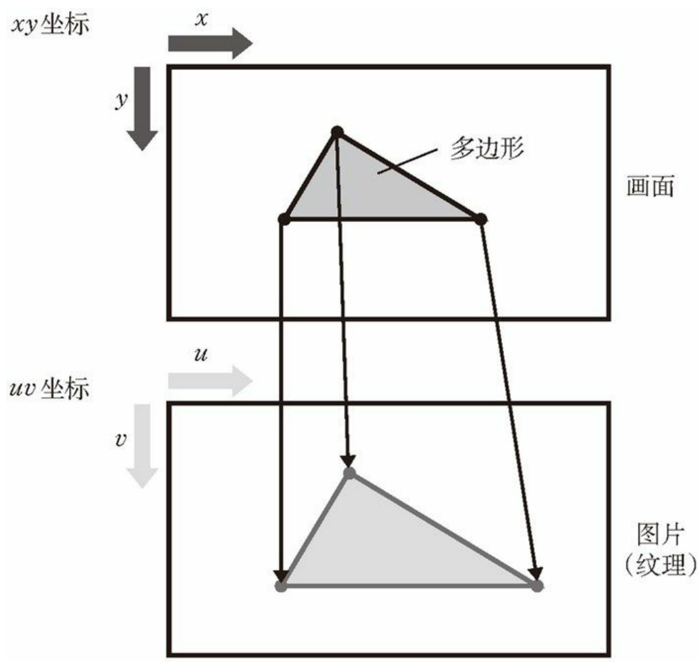


图 **5-1-2**　通过 ***uv*** 坐标实现多边形的顶点与图片中的坐标相对应

Draw2DPolygon 函数中也对 3 个顶点同时设置了画面上的显示坐标 *xy* 以及图片上的渲染坐标 *uv* 。

接下来就是示例程序 Wipe1_1.cpp 中最重要的部分：将 *xy* 坐标对应为 *uv* 坐标。具体来说，渲染两张图片需要 4 个多边形，每个多边形都有 3 个顶点坐标，这些顶点坐标可以通过以下计算得到。令边界线的 *x* 坐标为 b_x

，画面宽度为 w_D ，画面高度为 h_D ，如图 5-1-3 所示，有以下关系成立。

- 对于第 1 个多边形：

  渲染 xy 坐标为 $(0, 0)$、$(0, h_D)$、$(b_x, 0)$ 三点

  uv 坐标为 $(0, 0)$、$(0, 1)$、$(\frac{b_x}{w_D}, 0)$ 三点

- 对于第 2 个多边形：

  渲染 xy 坐标为 $(b_x, 0)$、$(0, h_D)$、(b_x, h_D) 三点

  uv 坐标为 $(\frac{b_x}{w_D}, 0)$ 、$(0, 1)$、$(\frac{b_x}{w_D}, 1)$ 三点

- 对于第 3 个多边形：

  渲染 xy 坐标为 $(b_x, 0)$、(b_x, h_D)、$(w_D, 0)$ 三点

  uv 坐标为 $(\frac{b_x}{w_D}, 0)$ 、$(\frac{b_x}{w_D}, 1)$ 、$(1, 0)$ 三点

- 对于第 4 个多边形：

  渲染 x_y 坐标为 $(w_D, 0)$、(b_x, h_D)、(w_D, h_D) 三点

  uv 坐标为 $(1, 0)$、$(\frac{b_x}{w_D}, 1)$ 、$(1, 1)$ 三点

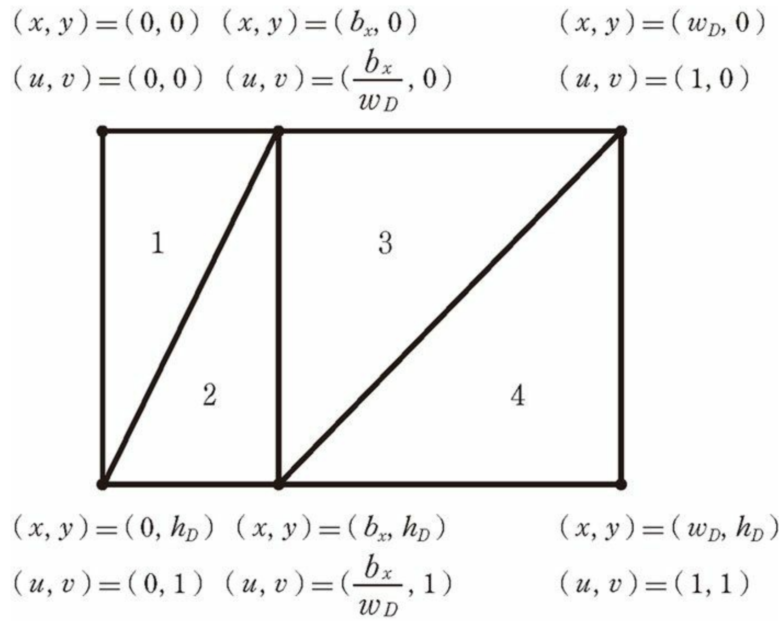


图 **5-1-3**　多边形与图片坐标及 ***uv*** 坐标

各顶点的 *xy* 坐标及 *uv* 坐标是有一定的对应关系的。通过图 5-1-3 可以看出这种关系为

$$u = \frac{x}{w_D}$$
$$v = \frac{y}{h_D}$$

这是因为对于画面宽度为 w_D 、画面高为 h_D 的图形来说，画面坐标所转换的 *uv* 坐标会将图形的 长宽映射到 0～1 的区间内。即图形左上角为 (0, 0)、右上角为 (1, 0)、左下角为 (0, 1)、右下角为 (1, 1)（图 5-1-4）。

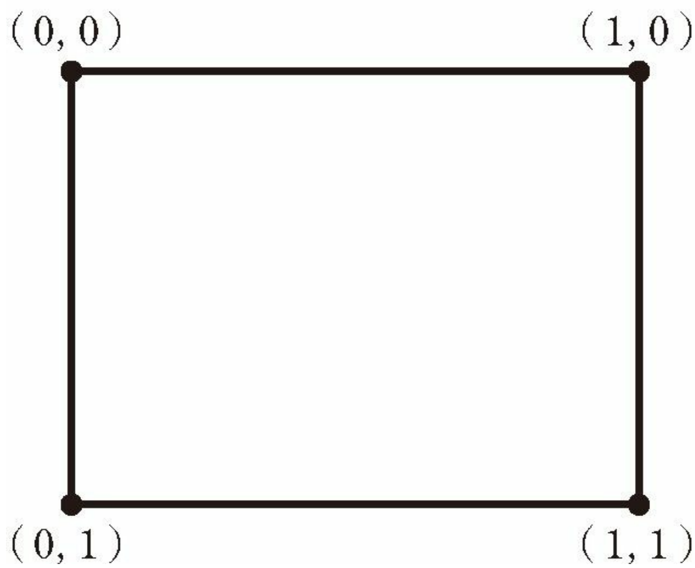


图 **5-1-4**　图片的 ***uv*** 坐标

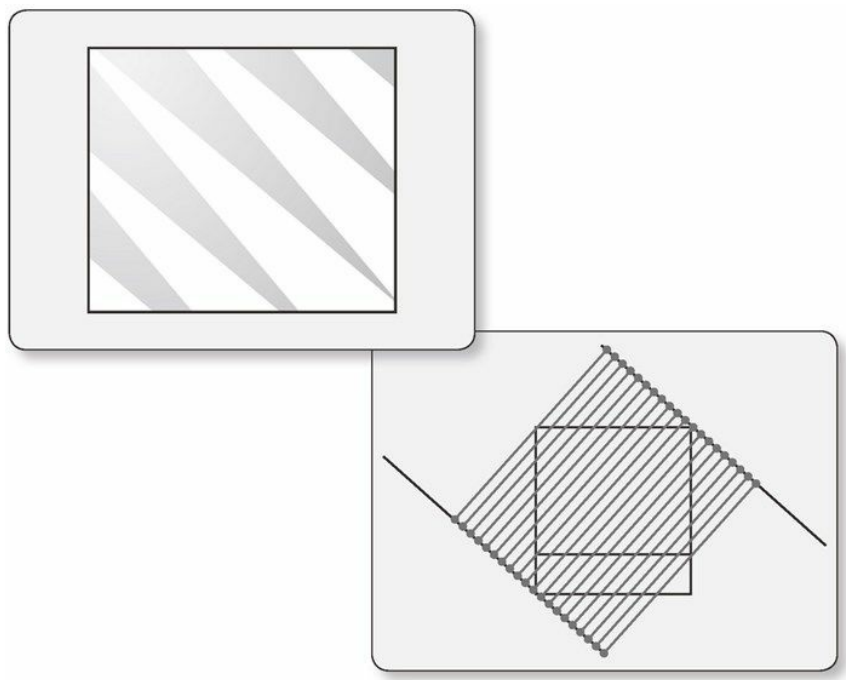
由于目前我们要实现的是画面切换效果，必须将画面的渲染坐标与所要渲染的图片位置对应起来。因此上例中为了使 x 坐标在 0～w_D 范围内变化时，u 坐标可以在 0～1 范围内变化，用 x 坐标除以画面宽度 w_D 得到了 u 坐标，同理用 y 坐标除以画面高度 h_D 得到了 v 坐标。如果对上 述说明仍不太理解，可以对 Draw2DPolygon 函数中渲染的多边形的 *uv* 坐标做各种修改来观察其变化。

此外，多边形的 *uv* 坐标之所以不以像素而以 0～1 表示图片整体的坐标，是因为这样处理后，即便更换了图片也仍然可以使用相同的 *uv* 坐标。也就是说，u_v 坐标并不受图片物理大小的影响，3D 渲染中有一种叫做 Mipmap 的贴图技巧就是基于这个特性实现的，可以在贴图过程中实时更换不同大小（或分辨率）的图片。

5.2 斜向扫描式画面切换

Key Word　向量形式的直线、剪裁

RANK/normal



改变画面切换效果的方向，可以让画面表现更加多样化。在水平方向切换的基础上，本小节让我们来一起学习斜方向的画面切换效果。

本小节将介绍斜向扫描式画面切换效果。本小节的内容是建立在5.1节的基础上的，如果对多边形等知识还不够熟悉，建议先掌握上一小节的内

容。



图 **5-2-1**　斜向扫描式画面切换程序

示例程序 Wipe_2_1.cpp 实现了斜方向的画面切换效果，程序中的重点是以下部分。

代码清单 **5-2-1**　进行斜方向画面切换的主要部分（**Wipe_2_1.cpp** 片段）

```
int DrawChangingPictures( void )
{
    int                 i;
    float               xt[3], yt[3];
    float               xb[3], yb[3];

    fBoundary_t += fBoundary_v;
    if ( fBoundary_t < -VIEW_WIDTH / 2.0f )
        fBoundary_t = -VIEW_WIDTH / 2.0f;
        fBoundary_v = -fBoundary_v;
    }
    if ( fBoundary_t > VIEW_HEIGHT / 2.0f )
        fBoundary_t = VIEW_HEIGHT / 2.0;
        fBoundary_v = -fBoundary_v;
    }
    xt[0] = VIEW_WIDTH - VIEW_WIDTH / 2.0f;
    yt[0] = 0.0f       - VIEW_WIDTH / 2.0f;
    xt[1] = VIEW_WIDTH + fBoundary_t;
```

```
        yt[1] = 0.0f       + fBoundary_t;
        xt[2] = VIEW_WIDTH + VIEW_HEIGHT / 2.0f;
        yt[2] = 0.0f       + VIEW_HEIGHT / 2.0f;
        for ( i = 0; i < 3; i++ ) {
            xb[i] = xt[i] - VIEW_WIDTH;
            yb[i] = yt[i] + VIEW_WIDTH;
        }
    // 以下省略
```

在上面的程序中，首先以向量形式作出了一条直线，并以此直线为运动轨道，使沿斜 45 度角切分画面的分界线在该轨道上运动。因此作为轨道的直线，也同样对画面呈 45 度角倾斜，并且通过点 (VIEW_WIDTH, 0)，即画面的右上角（图 5-2-2）。

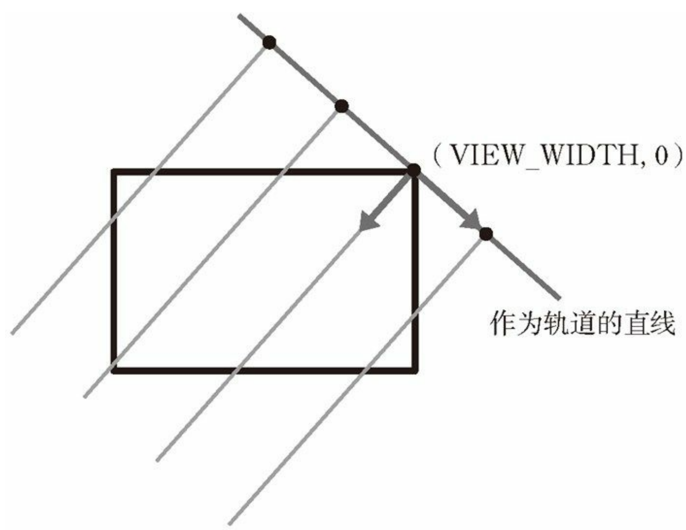


图 **5-2-2**　以向量形式表示将画面沿 **45** 度角切分的直线

将这条直线表示为向量方程，假设轨道直线上点的位置向量为 $\boldsymbol{p}$，则有

$$\boldsymbol{p} = \boldsymbol{a}\,t + \boldsymbol{b} \quad (\boldsymbol{a} = (1, 1)、\boldsymbol{b} = (\text{VIEW_WIDTH},0))$$

沿轨道直线向向量 (-1, 1) 的方向（即与轨道直线正交，且将画面沿斜 45 度角切分的方向）延伸得到的线，就是两张背景图片的分界线（参考图 5-2-2）。接下来我们就要考虑剩下的问题，包括这条分界线沿轨道直线应该有多大的运动范围，以及分界线应该有多长才能覆盖整个画面等。

- 分界线的运动范围

  首先来考虑分界线的运动范围。刚才得到的轨道直线的向量方程

  $\boldsymbol{p} = \boldsymbol{a}\,t + \boldsymbol{b}$ （$\boldsymbol{a}$ = (1, 1)、 $\boldsymbol{b}$ = (VIEW_WIDTH,0)）

  中，我们已经知道分界线延伸的方向，就是轨道直线上的一点到向量 (-1, 1) 的方向。因此当方程中 *t* 为最小值时，分界线正好通过画面左上角（参考图 5-2-3 左）。此时如图所示，轨道直线上点的 *x* 坐标为 VIEW_WIDTH/2，t=-VIEW_WIDTH/2。同理，当 *t* 为最大值时，分界线正好通过画面右下角（图 5-2-3 右），此时轨道直线上点的 *y* 坐标为 VIEW_HEIGHT/2，t=VIEW_HEIGHT/2。

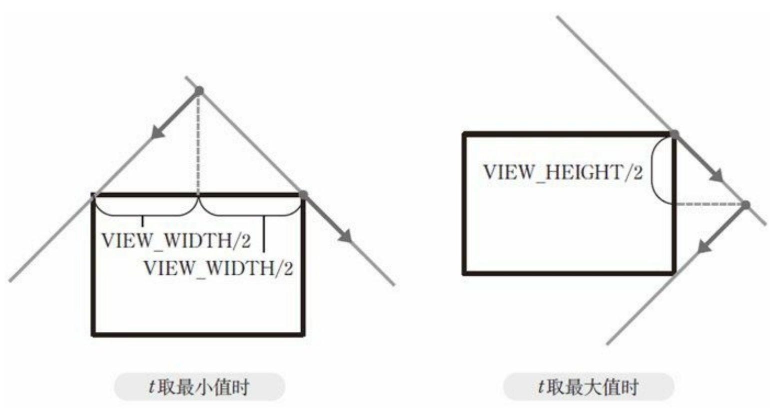


图 **5-2-3** ***t*** 的最大值与最小值

综上可知

-VIEW_WIDTH/2 ≤ t ≤ VIEW_HEIGHT/2

t 为最小值时分界线通过画面左上角，*t* 为最大值时分界线通过画面右下角。因此为了使 *t* 的上限及下限部分，分别与图片 A 的边缘及图片 B 的边缘相对应，程序中分别对①图片 A 的边缘（*t* =VIEW_WIDTH/2）、② 分界线（*t* =fBoundary_t）、③ 图片 B 的边缘（*t* =VIEW_HEIGHT/2）在直线上的位置坐标进行了计算，如下所示。

```
    xt[0] = VIEW_WIDTH - VIEW_WIDTH / 2
    yt[0] = 0.0f       - VIEW_WIDTH / 2
    xt[1] = VIEW_WIDTH + fBoundary_t;
    yt[1] = 0.0f       + fBoundary_t;
    xt[2] = VIEW_WIDTH + VIEW_HEIGHT /
    yt[2] = 0.0f       + VIEW_HEIGHT /
```

请注意这三个点都位于通过点 (VIEW_WIDTH, 0)、且与向量 (1, 1) 平行的直线上。

- 分界线的长度

接下来需要考虑分界线的长度。从最终效果看，我们希望分界线无论位于什么位置，都可以覆盖整个画面。为了让算法更简单，程序中我们并不会做一根正好覆盖画面的分界线，而是会制作一根比所需长度长很多的分界线。在渲染多边形时，超出画面外的部分会触发剪裁 操作被剪裁掉，比如，即便程序中指定在画面外的某个区域进行渲染，（除极端情况之外）也不会发现画面有什么

问题。因此在 Wipe_2_1.cpp 中，分界线的具体长度就是画面长宽均为 VIEW_WIDTH 时分界线的长度（参考图 5-2-4）。

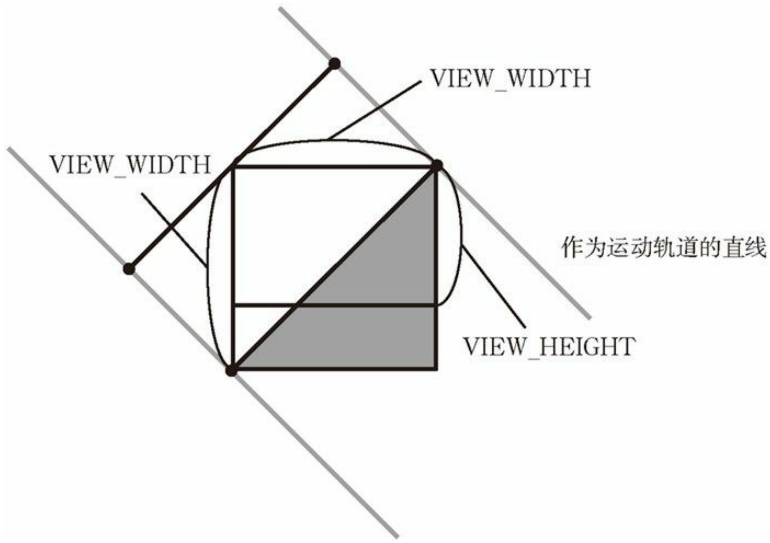


图 **5-2-4**　分界线的长度

如上图所示，此时分界线的长度，等于边长为 VIEW_WIDTH 的正方形对角线的长度。为了保证分界线的长度，需要让轨道直线上的点向 *x* 方向移动 -VIEW_WIDTH，向 *y* 方向移动 VIEW_WIDTH，并将分界线延伸到这里（参考图 5-2-4）。让我们来验证一下。当实际画面宽 VIEW_WIDTH 为 640、画面高 VIEW_HEIGHT 为 480 时，画面所需的分界线确实比边长为 VIEW_WIDTH 的正方形的分界线短，因此只需要保证这么长的分界线就可以了。上文中提到的让轨道直线上的点向 *x* 方向移动 -VIEW_WIDTH、向 *y* 方向移动 VIEW_WIDTH，并将分界线延伸到该处这一操作，在程序中体现为

```
    for ( i = 0; i < 3; i++ ) {
        xb[i] = xt[i] - VIEW_WIDTH;
        yb[i] = yt[i] + VIEW_WIDTH;
    }
```

这样我们就分别对图片 A 的边缘、分界线、图片 B 的边缘计算得到了相应的点的坐标，且为了保证能覆盖画面，各个点都偏离 (-1, 1) 方向足够远的距离。接下来就是使用计算得到的这 6 个点，分别使用两个三角多边形来渲染图片 A 与图片 B（参考图 5-2-5）。

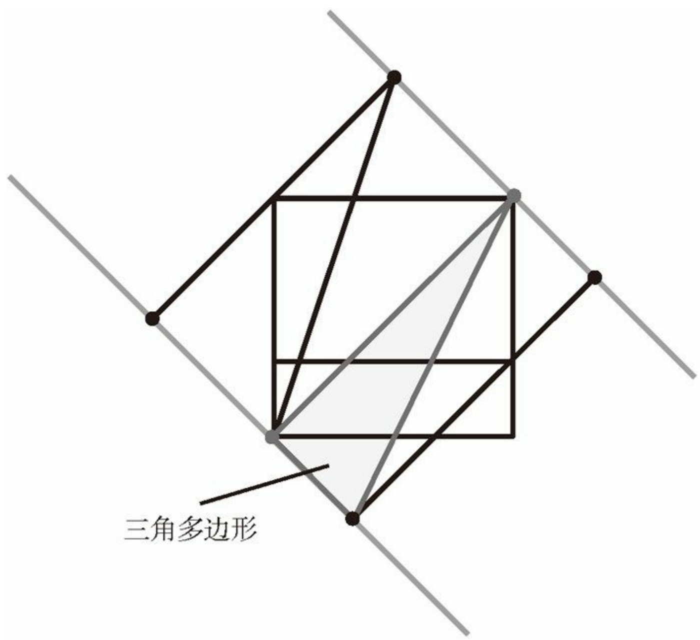


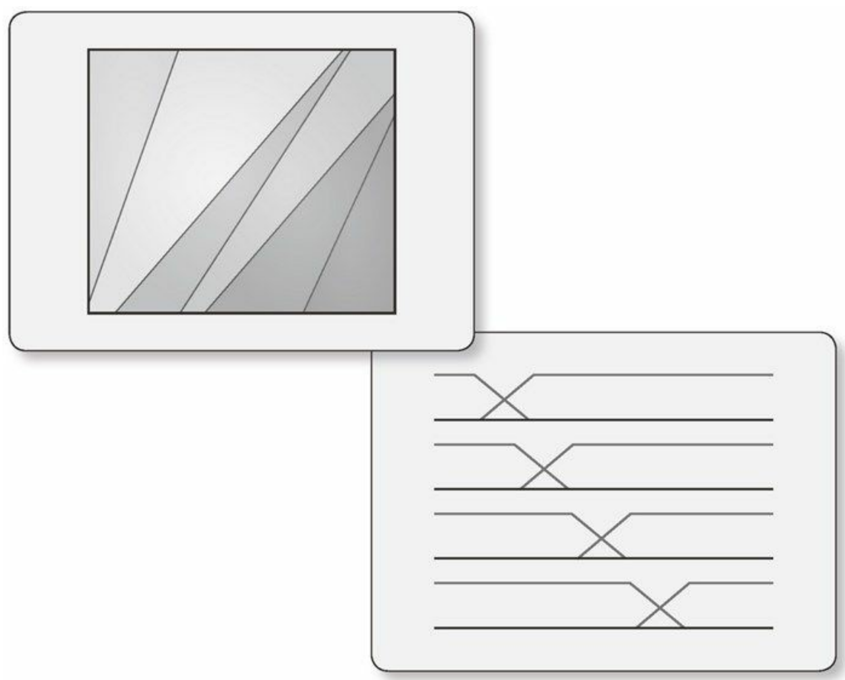
图 **5-2-5**　使用计算得到的 **6** 个点，分别使用两个三角多边形来渲染图片 **A** 与图片 **B**

类似这样使用斜向的分界线切分画面时，使用以向量形式表示的直线的方程式会非常方便。本小节只考虑了斜 45 度角，不过基于向量来考虑的话，不仅限于 45 度角，具有各种角度的分界线的画面切分实现起来也都比较简单。对此有兴趣的读者，可以自己尝试实现以积累更多经验。

5.3　使用带模糊效果的分界线进行画面切换

Key Word　渐变（**gradation**）、**Alpha** 合成（**Alpha blending**）

RANK/normal



使用两张图片进行画面切换时，如果分界线非常明显，会让过渡显得生硬，影响整体效果。对分界线增加模糊（也称为渐变）处理，可以让画面切换更加平滑，本小节就让我们一起来看下突现这一效果的方法。

本小节将介绍使用有模糊效果的分界线进行画面切换的方法。



图 **5-3-1**　使用模糊的分界线沿斜 **45** 度角进行画面切换的程序

示例程序 Wipe_3_1.cpp 实现了这一效果。程序中的大部分与 5.2 节中的示例程序 Wipe_2_1.cpp 一致，详细说明请参考上一节的内容。本小节主要介绍程序中与 Wipe_2_1.cpp 不同的部分。在上一节的 Wipe_2_1.cpp 中，画面被分为了两个区域，即图 A 区域及图 B 区域。本小节的 Wipe_3_1.cpp 中则会将画面分为 3 个区域，即图 A 区域、图 A 与图 B 的混合区域、图 B 区域（参考图 5-3-2）。

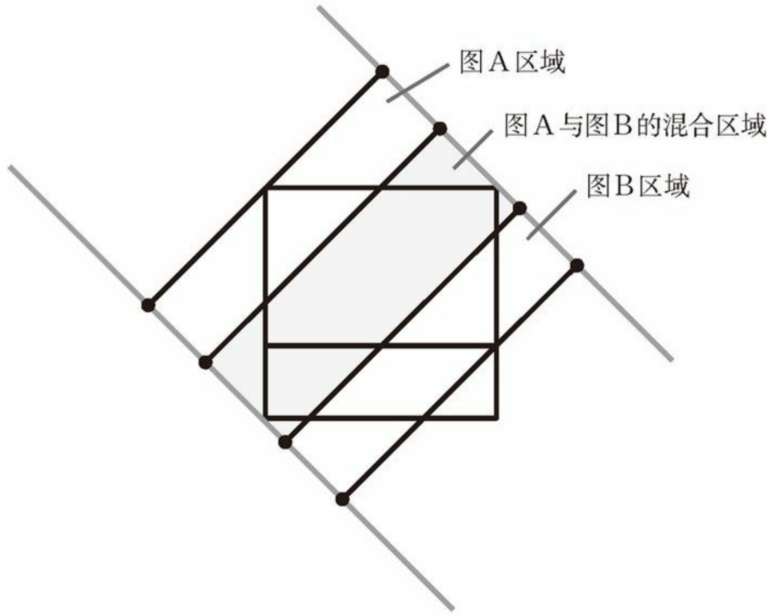


图 **5-3-2**　将画面分割为 **3** 个区域

为了将画面分成 3 个区域，需要有画面两端及内部的两条分界线，合计 4 条分界线，Wipe_3_1.cpp 中对 4 条分界线进行了如下设定。

代码清单 **5-3-1**　将画面分成 **3** 个区域的 **4** 条分界线的设定（**Wipe_3_1.cpp** 片段）

```
    xt[0] = VIEW_WIDTH - VIEW_WIDTH / 2.0f -
    yt[0] = 0.0f       - VIEW_WIDTH / 2.0f -
    xt[1] = VIEW_WIDTH + fBoundary_t;
    yt[1] = 0.0f       + fBoundary_t;
    xt[2] = VIEW_WIDTH + fBoundary_t + GRAD_
    yt[2] = 0.0f       + fBoundary_t + GRAD_
    xt[3] = VIEW_WIDTH + VIEW_HEIGHT / 2.0f
    yt[3] = 0.0f       + VIEW_HEIGHT / 2.0f
```

在上面的程序中，Wipe_2_1.cpp 所没有的是数组 [2] 部分（代码清单 5-3-1 ①）。相当于我们在原图 A 与图 B 的分界线（数组 [1] 的部分）的基础

上，在其向 *x* 方向及 *y* 方向同时偏离 GRAD_WIDTH 的位置，新增了一条分界线。由此可得到

- 区域 1：位于数组 [0] 与数组 [1] 之间（只渲染图 A）
- 区域 2：位于数组 [1] 与数组 [2] 之间（渲染图 A 与图 B 的半透明混合效果）
- 区域 3：位于数组 [2] 与数组 [3] 之间（只渲染图 B）

这样 3 个不同的区域。请注意上面控制渐变区域宽度的 GRAD_WIDTH 这个常数并不是以像素为单位的。因为同时在 *x* 方向及 *y* 方向都增加了 GRAD_WIDTH，所以如果实际渐变区域的宽度以像素为单位，则为 GRAD_WIDTH 的 $\sqrt{2}$ 倍，不是整数。另外在上面的程序中，数组 [0] 的分界线的 *xy* 坐标同时向负方向扩展了 GRAD_WIDTH。

```
    xt[0] = VIEW_WIDTH - VIEW_WIDTH / 2.0f -
    yt[0] = 0.0f       - VIEW_WIDTH / 2.0f -
```

由于在渐变部分完全超出画面外之前分界线会一直移动（即最终画面只保留一张图片），因此这样做才能确保在分界线移出画面后渐变部分没有残留在画面内。数组 [3]（画面右下的分界线）中也有同样的处理。

```
    xt[3] = VIEW_WIDTH + VIEW_HEIGHT / 2.0f
    yt[3] = 0.0f       + VIEW_HEIGHT / 2.0f
```

分界线的 *xy* 坐标同时向正方向（即向画面外的方向）扩展了 GRAD_WIDTH。

程序接下来进行了实际渲染，初看可能会觉得与之前的渲染方式不太一样。通常将一个区域分割为 3 份后，渲染应该按照以下顺序进行：在区域 1 渲染图 A →在区域 2 渲染图 A 与图 B 并进行半透明渐变处理→在区域 3 渲染图 B，总计 4 次渲染（如果是基于多边形渲染则需要 8 次）。而程序中实际只用了 3 次渲染（基于多边形则为 6 次）。这是为了让 2D 渲染系统能够进行高速渲染而进行的优化。因为类似半透明渲染这种消耗资源的操作应该尽可能地避免。在进行半透明渲染时，首先会读取画面（帧缓冲区）中已经渲染过的内容，然后对要渲染的区域进行一些必要的运算，最后还要将运算结果重新写回到画面上，这样的一次半透明渲染相比普通渲染，至少会多出一次对已有图像数据的读取操作，因此应当尽可能地回避。

> **POINT**　半透明渲染会读取帧缓冲区中已有的内容，花费更多运算时间，应当以最低限度使用。

在区域 2，即对图片 A 与图片 B 进行半透明混合的部分，其实不需要对图 A 与图 B 都进行半透明渲染。只需要对图 A 进行普通渲染，并在渲染图 B 时对其进行一次 **Alpha** 合成，就能很好地实现图 A 与图 B 的半透明混合效果。如果更进一步来考虑，对于区域 2 中渲染的图 A，以及区域 1 中渲染的图 A，就没有必要分别处理。因为区域 1 与区域 2 本来就是相连的，图 A 又位

于两个区域中，那么将两个区域连起来对图 A 进行 1 次渲染，就可以有效减少 1 次渲染指令。虽然这样的优化可能微不足道，但是只有坚持这种理念，才能让大型系统也保持超速通畅。也就是说，Wipe_3_1.cpp 最终是由以下步骤组成的。

① 将区域 1 与区域 2 整合为一个区域，对图 A 进行普通渲染

② 在区域 3 中对图 B 进行普通渲染

③ 在区域 2 中对图 B 进行半透明渲染

通过总计 3 个步骤完成了整个渲染过程（图 5-3-3）。当然以上都是以 2D 渲染系统为前提而使用的优化手段，如果是以 3D 为主的系统，通过使用多重纹理混合（multiple texture）技术，也可以高速地得到同样的渲染结果。具有 3D 系统相关知识的朋友，可以自己尝试使用多重纹理混合实现上面的渲染过程。

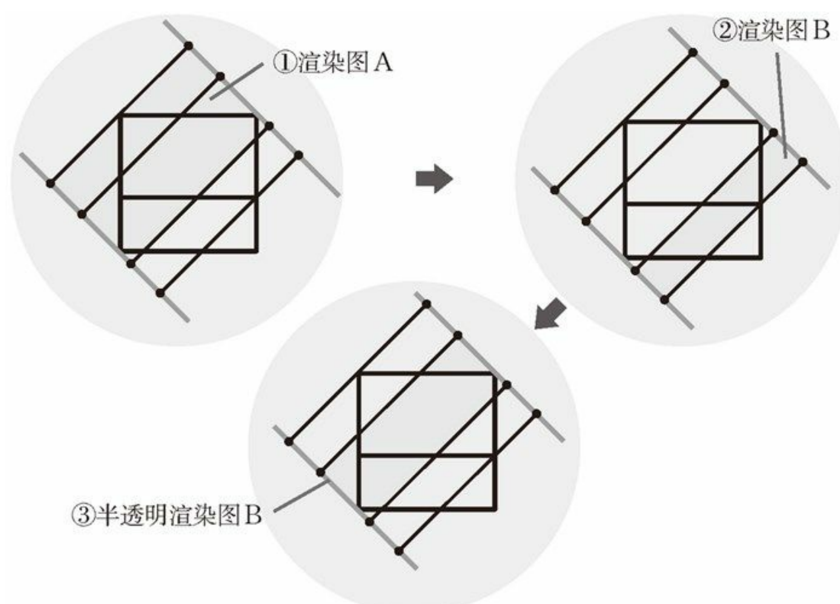
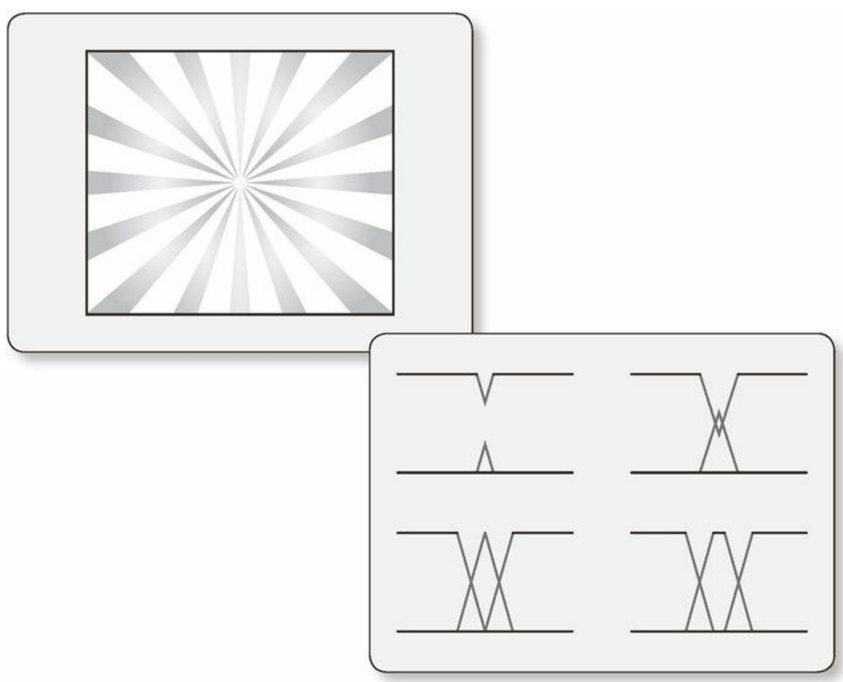


图 **5-3-3**　渲染的 **3** 个步骤

5.4 使用圆形进行画面切换

Key Word　避免重复渲染、环形、α 值

RANK/hard



用于画面切换的分界线，并不是只有直线。在动画作品中经常能看到使用圆形作为分界线进行画面切换。在本小节，我们就来学习这种效果的实现。

本小节将介绍使用圆形作为分界线进行画面切换。



图 **5-4-1**　使用圆形分界线进行画面切换的程序

示例程序 Wipe_4_1.cpp 实现了圆形的画面切换效果。程序中的主要部分如下所示。

代码清单 **5-4-1**　使用圆形分界线进行画面切换的主要程序（**Wipe_4_1.cpp** 片段）

```
int InitChangingPictures( void )
{
    fBoundary_r1 = MIN_R;
    fBoundary_r2 = sqrtf( VIEW_WIDTH * VIEW_
    fBoundary_v = 5.0f;

    return 0;
}


int DrawChangingPictures( void )
{
    int                     i;
    float               fAngle1, fAngle2;
    float               fAngleDelta;
    float               xt[4], yt[4];
    fBoundary_r1 += fBoundary_v;
    if ( fBoundary_r1 < MIN_R ) {
        fBoundary_r1 = MIN_R;
        fBoundary_v = -fBoundary_v;
```

```
    }
    if ( fBoundary_r1 > fBoundary_r2 ) {
        fBoundary_r1 = fBoundary_r2;
        fBoundary_v = -fBoundary_v;
    }
    fAngleDelta = 2.0f * PI / DIVIDE_NUM;
    fAngle1 = 0.0f;
    fAngle2 = fAngleDelta;
    for ( i = 0; i < DIVIDE_NUM; i++ ) {
        xt[0] = VIEW_WIDTH  / 2.0f + fBounda
        yt[0] = VIEW_HEIGHT / 2.0f + fBounda
        xt[1] = VIEW_WIDTH  / 2.0f + fBounda
        yt[1] = VIEW_HEIGHT / 2.0f + fBounda
        xt[2] = VIEW_WIDTH  / 2.0f + fBounda
        yt[2] = VIEW_HEIGHT / 2.0f + fBounda
        xt[3] = VIEW_WIDTH  / 2.0f + fBounda
        yt[3] = VIEW_HEIGHT / 2.0f + fBounda
//以下省略
```

程序中使用正弦、余弦制作了以画面正中央（VIEW_WIDTH/2，VIEW_HEIGHT/2）为圆心的两个圆。一个圆的半径为 fBoundary_r1，另一个圆的半径为 fBoundary_r2。运行 Wipe_4_1.cpp 就可以看到，以画面中心为圆心的圆会自己放大缩小，并作为分界线切换了两张图片的显示。圆形的分界线明明只有一条，程序中为什么要创建出两个不同半径的圆呢？其实这是为了避免两张图片被重复渲染。假设画面正中央的图片为图 A，其外围的图片为图 B，如果允许两张图片重复渲染，那么只要在整画面范围内渲染图 B，然后在圆形范围内渲染图 A 就可以得到正确结果。但是这样一来渲染图 A 的区域一定会重复渲染图 A 与图 B，特别是当图 A 几乎覆盖整个画面时，重复渲染的面积将非常大，从而造成大量的运算被白白浪费。因此 Wipe_4_1.cpp 中准备了半径不同的圆 1（半径为 fBoundary_r1）与圆 2（半径为 fBoundary_r2）， 其中，

- 图 A 只在圆 1 的区域中渲染
- 图 B 只在圆 1 与圆 2 之间的环形区域内渲染

这样就避免了图片的重复渲染。

> **POINT**　重复渲染面积很大时会造成运算的严重浪费，为此准备两个圆，一个覆盖整个画面，另一个在内部渲染其他图片，就可以避免重复渲染的问题。

在上述情况下，外侧的圆 2 的半径就至少要保证圆能覆盖整个画面。此时如果对圆 2 进行普通渲染，由于超出画面的部分会被剪裁掉，因此圆的半径不需要正好覆盖画面，比这再大一些也没什么问题（但也不能无限制的大，过大的数字运算会产生计算误差，可能让画面出现奇怪的问题）。在上例中，为了使当圆 1 与圆 2 一致时，正好只有图 A 可以覆盖整个画面，将圆 2 的半径设置为了勉强可以覆盖整个画面。所谓勉强可以覆盖整个画面的半径，就是从画面正中央的坐标，到画面四个角中的一个角的距离。使用勾股定理可以求得这一距离，程序中计算圆 2 的半径的是 InitChangingPicture 函数内的

```
    fBoundary_r2 = sqrtf( VIEW_WIDTH * VIEW_
```

这一部分。很容易可以看出，画面正中央到任意一角的距离，等于画面对角线长的一半（参考图 5-4-2）。

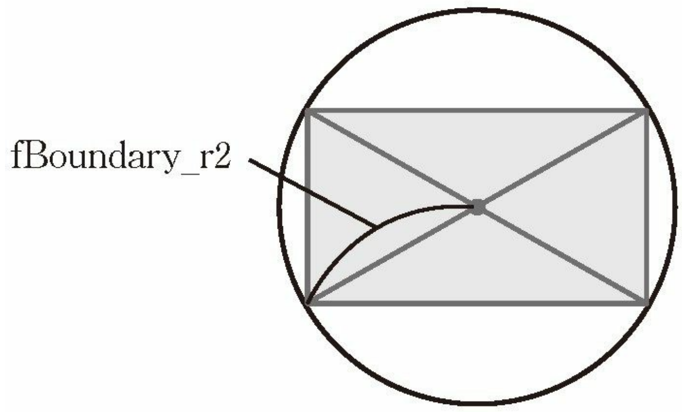


图 **5-4-2**　使用勾股定理计算画面正中央到一角的距离

- 使用带模糊效果的圆形分界线进行画面切换

  接下来将介绍分界线为圆形，并且分界线带有模糊效果的画面切换。示例程序 Wipe_4_1a.cpp 实现了这一效果。由于程序源代码很长，这里就不再直接引用代码，仅就代码中使用的算法进行说明（源代码下载地址可以参考文前的“关于本书”）。

  这个程序略显复杂主要还是因为程序中存在很多条件分支。关于斜向模糊的分界线的实现，我们在前一小节中已经进行过说明，即需要准备 4 条分界线，并分别渲染图 A 区域、图 A 与图 B 的混合区域、图 B 区域总计 3 个区域。其实当分界线为圆形时，也可以采用同样的思路，将画面分为 3 个区域分别渲染，但有所不同的是，只渲染图 A 的区域（不管是画面内还是画面外）并不是一定存在的。运行 Wipe_4_1.cpp 可以看到，首先图 A 与图 B 的混合区域会出现在画面正中央，当该区域的半径超过渐变区域的宽度时，只渲染图 A 的区域才会出现。也就是说，根据渲染图 A 的区域存在与否，整体渲染策略会有所不同。具体来说，只渲染

图 A 的区域不存在时，当然就没有必要渲染图 A，此时图 A 与图 B 的混合区域为圆形；另一方面，如果存在只渲染图 A 的区域，则程序需要渲染图 A，此时渲染图 A 的区域为圆形，而渲染图 A 与图 B 的混合区域就会变为环形（图 5-4-3）。

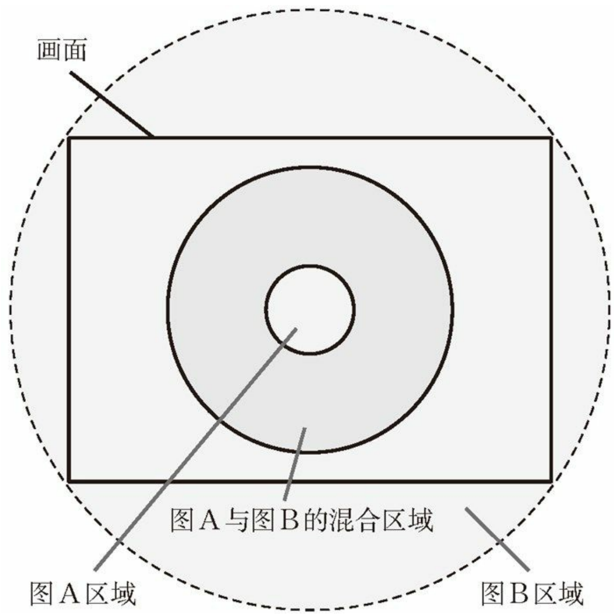


图 **5-4-3**　渲染的区域和内容

可以确定的是无论上述哪种情况，只渲染图 B 的区域都为环形（只是外侧的圆在画面外）。由于只渲染图 A 的区域存在与否对处理方式有较大影响，因此在 Wipe_4_1a.cpp 中，准备了 3 个半径 fBoundary_r1、fBoundary_r2、fBoundary_r3，具体作用如下。

- fBoundary_r1 : 只渲染图 A 的区域与图 A 图 B 混合区域的分界线的半径
- fBoundary_r2 : 图 A 图 B 混合区域与只渲染图 B 的区域的分界线的半径

- fBoundary_r3 : 只渲染图 B 的区域的外侧的分界线的半径（定值）

这里需要注意的是，fBoundary_r1 在只渲染图 A 的区域不存在时是没有意义的。同时从数值来看，fBoundary_r1 应当始终比 fBoundary_r2 的值小 GRAD_WIDTH（即渐变区域的宽度）。而由于 fBoundary_r2 是从 0 开始变化的，因此 fBoundary_r1 可能取负值（当 fBoundary_r2 为 0 时 fBoundary_r1 为 -GRAD_WIDTH）。当然半径为负值的图形是不可能存在的，fBoundary_r1 为负值，即代表只渲染图 A 的区域不存在。因此检查只渲染图 A 的区域是否存在，只需要在 if 语句中判断 fBoundary_r1 的值就可以了。但是最终的判断条件并不是 fBoundary_r1 是否为负，因为即使 fBoundary_r1 不为负值，如果半径小于 1，也会因渲染区域过小而认为渲染区域不存在。

- 只渲染图 **A** 的区域不存在时的处理

上面介绍了 Wipe_4_1a.cpp 的思路，但是还遗留了一个很大的问题。那就是在只渲染图 A 的区域不存在的情况下，渲染图 A 与图 B 的混合区域时，图 A 与图 B 的混合比率应该设定为多少的问题。比如当 fBoundary_r2 接近于 0 时，也就是图 A（以半透明形式）刚刚开始显示时，即使是图 A 与图 B 的混合区域的中心部分，也只会对图 A 做最小限度的渲染。当图 A 与图 B 的混合区域半径逐渐增大到接近渐变区域的宽度时，中心部分中图 A 的渲染程度就很高了。而当半径超过渐变区域的宽度时，中心部分将只渲染图 A。在前一小节中，我们分别准备了只渲染图 A 的点与只渲染图 B 的点，然后通

过分别移动这些点（不包括在画面外的移动）实现了混合区域的移动，但是这样的处理不适用于现在的情况，因为当只渲染图 A 的区域不存在时，还需要对画面正中央的点的透明度进行适当的控制。执行半透明控制的代码是 Wipe_4_1a.cpp 的以下部分。

代码清单 **5-4-2**　控制透明度渲染的部分（**Wipe_4_1a.cpp** 片段）

```
137  |            nCenterColor = ( ( int )( f
```

这行代码对画面正中央的点的颜色（*α* 值 = 包含透明度的数值）进行了设置。首先正中央的颜色（不含 *α* 值）为 0xffffff，这表示 R=255、G=255、B=255，RGB 都为最大亮度，即白色。*α* 值通过

```
( fBoundary_r2 * 255 / GRAD_WIDTH )
```

计算得到，*α* 值与颜色值一样，也是一个 0～255 的数值，为 0 时表示完全透明，为 255 时表示完全不透明。在上面的等式中，

- 当 fBoundary_r2 为 0 时，*α* 值也为 0

- 当 fBoundary_r2 为 GRAD_WIDTH，即与渐变区域相等时，*α* 值为 255

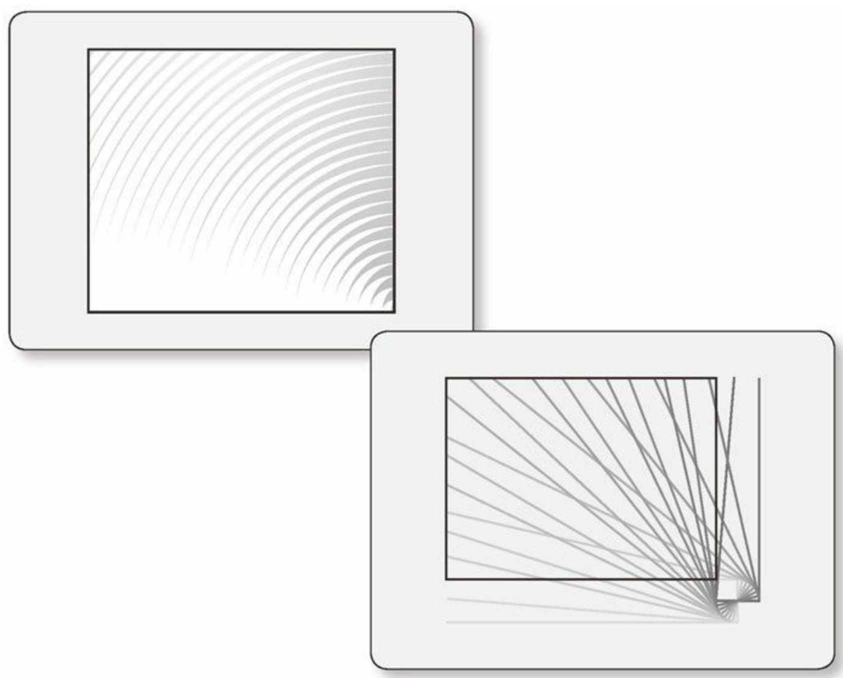
相当于对 α 值始终乘以一个变化的系数。如果对此不太理解，可以将程序中的 fBoundary_r2 设置为 GRAD_WIDTH，来验证一下此时的 α 值是否为 255。总之通过这种方式，就实现了 fBoundary_r2 为 0 时画面正中的图 A 完全透明，fBoundary_r2 等于渐变区域宽度时，画面正中的图 A 完全不透明（即只渲染图 A）这一效果。

本小节实现了新的画面切换效果，让图片的起始位置不再只限于画面边缘，但是这就需要对分界线做很多特殊处理。对此感兴趣的朋友，还可以挑战其他更多有趣的画面切换方式。

5.5　雨刷式画面切换

Key Word　避免条件分支

RANK/hard



在本小节，我们将学习以一点为中心，旋转分界线实现的画面切换。

本小节将讲解以画面一角为中心，以一条旋转的直线为分界线的画面切换效果。

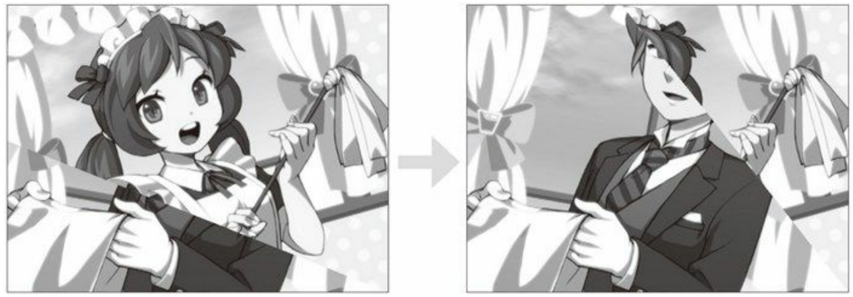


图 **5-5-1**　以画面一角为中心旋转分界线实现画面切换的程序

示例程序Wipe_5_1.cpp实现了雨刷式画面切换效果。程序中重要的部分是 DrawChangingPictures 函数的以下内容。

代码清单 **5-5-1**　以画面一角为中心旋转分界线进行画面切换的程序的主要部分（**Wipe_5_1.cpp** 片段）

```
    xt = VIEW_WIDTH  + DRAW_R * cosf( 3.0f *
    yt = VIEW_HEIGHT + DRAW_R * sinf( 3.0f *
    Draw2DPolygon( VIEW_WIDTH, VIEW_HEIGHT,
                   xt, yt, xt / VIEW_WIDTH,
                   -VIEW_HEIGHT, VIEW_HEIGHT
                   &g_tPic1 );
    Draw2DPolygon( VIEW_WIDTH, VIEW_HEIGHT,
                   VIEW_WIDTH, -VIEW_WIDTH,
                   xt, yt, xt / VIEW_WIDTH,
                   &g_tPic2 );
```

为了制作一条通过画面右下角并且可以旋转的分界线，首先需要准备一个以画面右下角 (VIEW_WIDTH, VIEW_HEIGHT) 为中心进行圆周运动的点，这一点可以通过以下方式制作。

```
    xt = VIEW_WIDTH  + DRAW_R * cosf( 3.0f *
```

```
073  |     yt = VIEW_HEIGHT + DRAW_R * sinf( 3.0f *
```

以画面右下角为中心时，画面将在中心点的左上方，那么将旋转的点 (xt, yt) 与画面右下角连接而成的分界线，在什么样的角度范围内能通过画面呢？在上例中，由于计算机画面中以 y 轴下方为正，因此当角度在 $\pi \sim \frac{3}{2}\pi$ 的范围内时，分界线通过画面（图 5-5-2）。

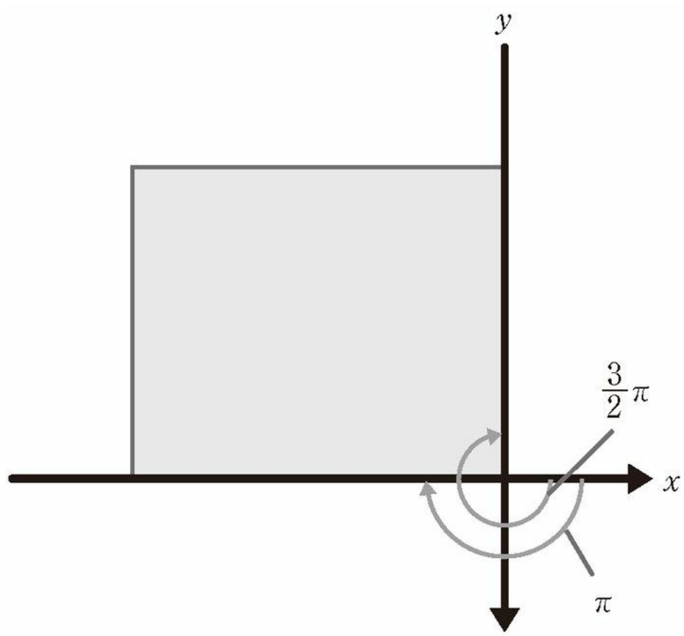


图 **5-5-2**　通过画面的分界线

在上面的两行程序中，传入正弦余弦函数的角度为 (3.0f * PI / 2.0 f - fAngle1) 就是出于这个原因。让等式中的 fAngle1 在0～ $\frac{\pi}{2}$ 的范围内变化，那么传入正弦余弦的值的范围就等于 $\pi \sim \frac{3}{2}\pi$，并且当 fAngle1 为 0 时，传入正弦余弦的角

度为 $\frac{3}{2}\pi$，即分界线正好与画面右端重合（图 5-5-2）。这样很容易就实现了分界线从画面右端开始旋转着出现的效果。

接下来要计算分界线所需的长度，稍微有点复杂。如果只考虑分界线，那么为了使分界线始终穿过画面，只要让分界线等于画面对角线长就足够了。但是如果我们需要根据分界线及画面边线，使用三角多边形对图 A 与图 B 进行渲染，条件分支就会变得有些复杂。

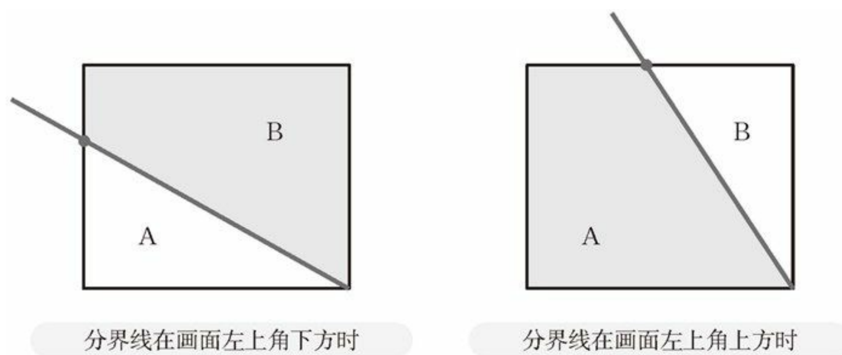


图 **5-5-3**　分界线在画面左上角下方（左图）及下方（右图）时

具体来说，如图 5-5-3 左所示，当分界线在画面左上角下方时，图上的区域 A 为三角形，区域 B 为四角形。如图 5-5-3 右所示，当分界线在画面左上角上方时，图上的区域 A 为四角形，区域 B 为三角形。当分界线正好通过画面左上角时，区域 A 区域 B 均为三角形。由于大部分系统中多边形只能为三角形，因此就不得不将四角形的区域再分割为两个三角形。这样一来究竟每个区域要渲染多少个多边形呢？就需要根据情况分别处理，程序也会变得复杂。

对此，Wipe_5_1.cpp 中并没有采用条件分支的处理方式，而是在多边形的大小上做文章，最终让每个区域只需要渲染一个三角多边形。具体如何

实现呢？首先，当分界线与画面右端重合时，渲染图 A 的三角多边形将覆盖整个画面；当分界线与画面下端重合时，则由渲染图 B 的三角多边形覆盖整个画面。除此以外的情况，即分界线位于中间时，就必须由图 A 的三角多边形与图 B 的三角多边形共同覆盖整个画面。按照这样的思路处理，我们就只需要知道多边形各顶点的位置就可以了。

首先从图 A 开始考虑。由于图 A 在分界线的下方渲染，因此就要在分界线与画面下边线之间的区域进行渲染。这时即使分界线与画面右端一致，三角多边形也必须能覆盖整个画面，为了实现这一点，最简单的方式就是让多边形的斜边为 45 度角（图 5-5-4）。

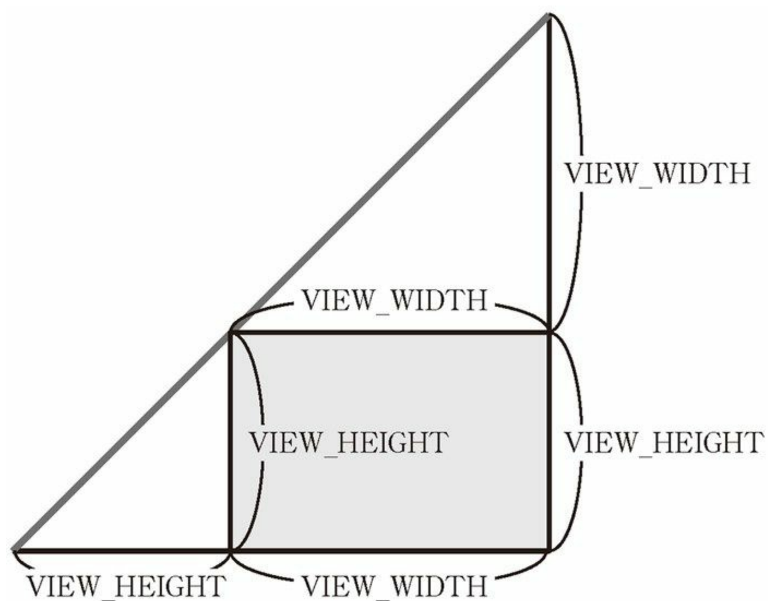


图 **5-5-4**　多边形的斜边为 **45** 度角

此时，为了使 45 度角的斜边正好通过画面左上角，就需要分界线的长为 VIEW_WIDTH + VIEW_HEIGHT，分界线的一个端点在画面下端，且其 *x* 坐标为 -VIEW_HEIGHT。也就是说，渲染图 A 所需的三角多边形的顶点为：(VIEW_WIDTH, VIEW_HEIGHT)，即画面右下

角；(-VIEW_HEIGHT, VIEW_HEIGHT)，即画面下端的分界线的一个端点。剩下的一个顶点就是图 A 与图 B 之间的分界线的另一个端点，位于以画面右下角为圆心、半径为 VIEW_WIDTH + VIEW_HEIGHT 的圆周上。该点在程序中为 (*xt* , *yt*)，可对应到下面的代码。

```
    Draw2DPolygon( VIEW_WIDTH, VIEW_HEIGHT,
                   xt, yt, xt / VIEW_WIDTH,
                   -VIEW_HEIGHT, VIEW_HEIGHT
                   &g_tPic1 );
```

接下来考虑图 B 的情况。图 B 在分界线的上方，因此就要在分界线与画面右端之间的区域内渲染。为了使分界线与画面下端重合时三角多边形也能覆盖整个画面，同样需要斜边为 45 度角，并且斜边会通过画面左上角，因此分界线长度与图 A 的情况一致，为 VIEW_WIDTH + VIEW_HEIGHT，分界线在画面右端的端点的 *y* 坐标为 -VIEW_WIDTH（图 5-5-4）。也就是说三角多边形的顶点分别为：画面右下角 (VIEW_WIDTH, VIEW_HEIGHT)、画面右边线上分界线的端点 (VIEW_WIDTH, -VIEW_WIDTH)，以及图 A 与图 B 间的分界线的另 一个端点。对应下面的代码。

```
    Draw2DPolygon( VIEW_WIDTH, VIEW_HEIGHT,
                   VIEW_WIDTH, -VIEW_WIDTH,
                   xt, yt, xt / VIEW_WIDTH,
                   &g_tPic2 );
```

- 使用带渐变效果的分界线进行画面切换

  接下来让我们对这个类似雨刷的分界线加上有透明度的渐变效果。示例程序 Wipe_5_1a.cpp 实现了这一效果。与 Wipe_5_1.cpp 相比，Wipe_5_1a.cpp 无疑更加复杂，代码行数也增加了不少，因此本文中不再引用代码，仅对实现的原理进行一些说明（源代码下载地址请参考文前的“关于本书”）。上面我们已经实现了使分界线像雨刷一样动作，为了让分界线还能有渐变效果，需要注意以下两个关键点。

  ① 如何实现当分界线的角度改变时渐变区域的宽度也是固定的。

  ② 分界线旋转的轴心位置。

  首先来看如何固定渐变区域的宽度这一点。在 4.2 节中，我们实现了无论光线投射向任何角度，其宽度都是固定的，而本小节就可以采用同样的处理方法。即作一条与分界线正交的向量 (vsx, vsy)，然后对其加减分界线两端的位置向量，进而得到固定宽度的分界线（参考图 4-2-2）。但是与光线处理不同的是，分界线的角度是已知的，因此只需要对分界线的角度减去 $\frac{\pi}{2}$ ，就可以得到与分界线垂直的向量。

  关于旋转的轴心位置，则需要好好考虑一下。当分界线没有渐变效果时，将轴心位置放在画面右下角是没有问题的。但是当分界线携带了一定宽度的渐变区域后，如果仍然将轴心设置在画面右下角，那么当分界线位于初始状态，比如垂直或水平时，就会有一半渐变区域遗留在画面内，之后无论分界线以何种角度运动，渐变区域也无法从画面中

移除（图 5-5-5）。

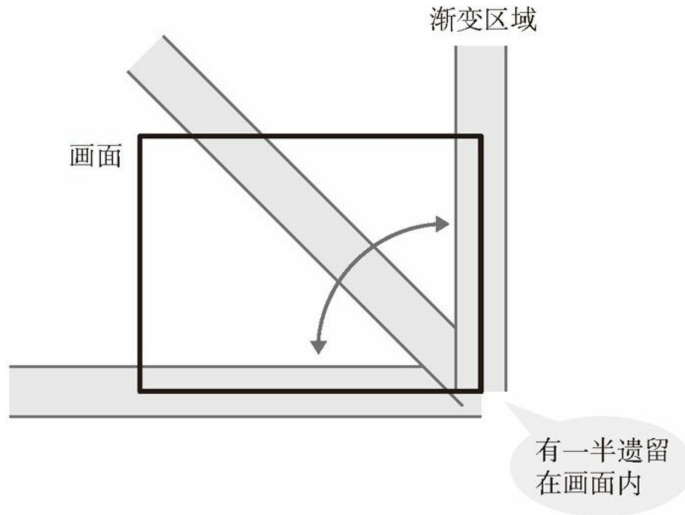


图 **5-5-5**　渐变区域的一半遗留在画面中无法移除

要解决这个问题，需要让分界线在水平或垂直状态时，渐变区域完全位于画面之外。为此应当在画面右下角的基础上，同时向右方及下方移动渐变区域宽度的一半，以这个画面外的点作为分界线旋转的轴心（图 5-5-6）。

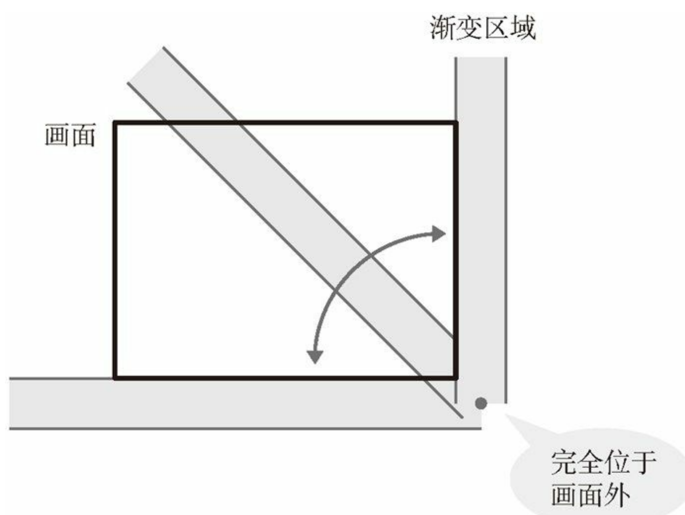


图 **5-5-6**　设置轴心位置以保证渐变区域完

全位于画面之外

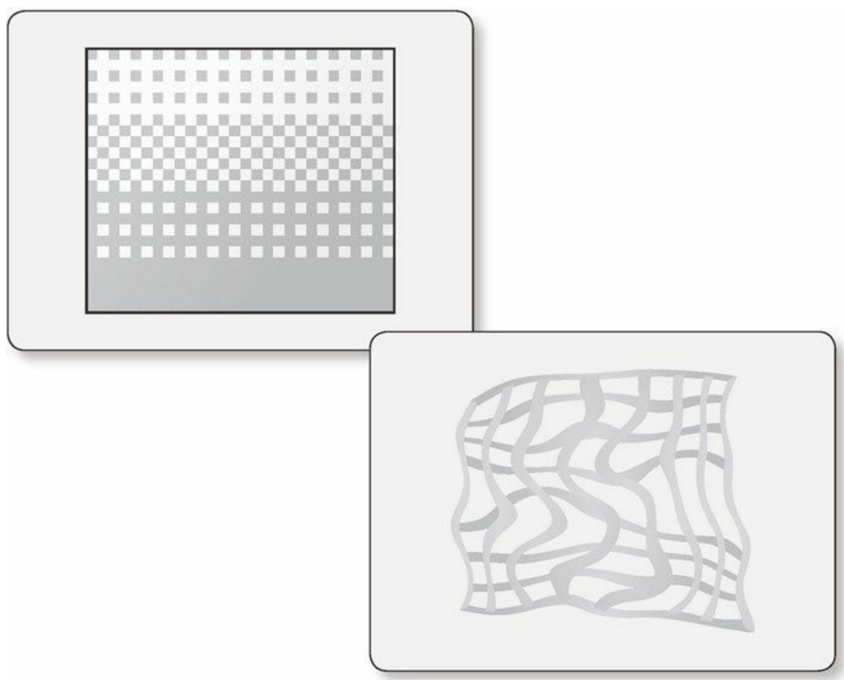
因此 Wipe_5_1a.cpp 中将分界线旋转的轴心位置设置为了 (VIEW_WIDTH + GRAD_WIDTH/2, VIEW_HEIGHT + GRAD_WIDTH/2)。由于现在分界线自身有宽度，因此多边形顶点不能直接放在分界线的轴心位置上。分界线的旋转轴心，也就是分界线两个端点中的一个，将其坐标加减与分界线正交的向量的坐标 (vsx, vsy)，就得到了多边形的顶点坐标。

还有一些细节需要注意，因为渐变区域的宽度问题需要将分界线的轴心放在画面外，此时也需要对分界线的长度相应地做一些延长，否则无论图 A 还是图 B，都无法只用一个多边形渲染覆盖整个画面。在 Wipe_5_1a.cpp 中，多边形的顶点超出画面的长度正好等于渐变区域的宽度，所以只需要将分界线延长渐变区域宽度的一倍，就可以保证三角多边形覆盖整个画面。

5.6 [进阶] 多种多样的画面切换方法

Key Word 遮罩图案、可编程着色器（**programmable shader**）、高斯滤波器（**gaussian filter**）

RANK/very hard



在本章最后，我们将结合实际应用的例子，对更复杂的画面切换方法进行一些介绍。

本小节将对游戏中经常使用的，以及过去比较流行的画面切换效果进行一些介绍。在游戏中，场

景更改时往往伴随着画面切换效果，比如从标题画面进入游戏画面时，以及 RPG 游戏中遭遇敌人（踩地雷式）时等。时至 2013 年，电脑游戏已经发展了 50 多年，画面切换始终伴随着电子游戏的发展而进行着各种尝试，现在的画面切换效果种类繁多，有各种各样的实现方法。

在电脑游戏发展初期，硬件制约异常严重，能实现的画面切换效果并不多。比如本书中介绍的以角色为单位，只更新画面一部分的方法。再比如由于当时无法更改颜色的亮度，因此就通过将比较亮的颜色按顺序切换到比较暗的颜色（如白→黄→紫→红→蓝等），来模拟类似颜色淡出的效果（画面由亮慢慢变暗）。后来计算机的性能略有提升，开始可以通过控制图片来完成一些新的效果，比如旧的画面卷动到画面外后，新的画面再卷动进入画面的效果。另外遮罩图案（mask pattern）也比较常用，即先准备好一个像素单位的图案来控制像素内是否对图片进行渲染，然后只需在单位面积内增减像素图案的数量，就可以模拟出类似淡出 / 淡入的效果了（图 5-6-1）。

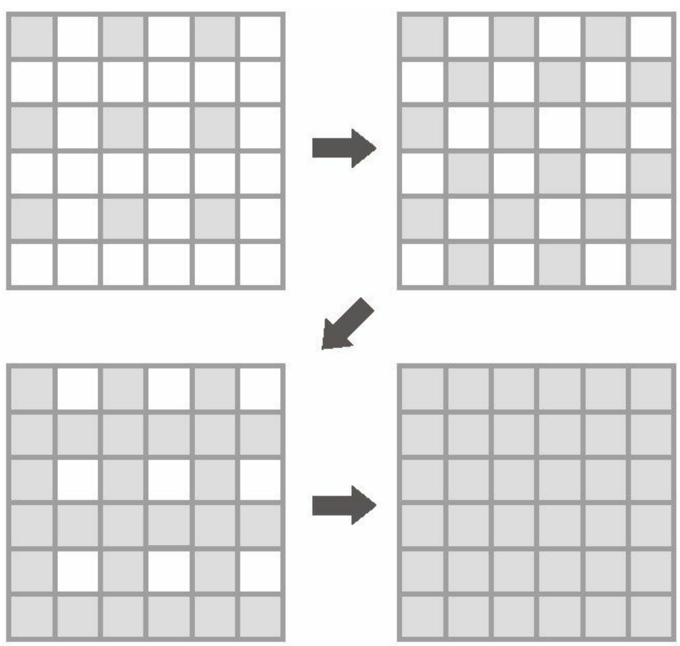


图 **5-6-1**　使用遮罩图案的画面切换效果

在那个时代，所有的画面切换效果的制作，只要懂得一定的编程知识就都不会遇到太大问题，如果还有一定的数学基础，就更加不在话下了。但是之后计算机的性能不断提升，半透明渲染、图像放大、缩小、旋转，以及对渲染完毕的图像进行复用等，都慢慢变为了基础功能，与此同时，对数学水平的要求也越来越高。比如在大部分半透明渲染中，如果想比较好地控制输出结果，就需要一定程度的数学知识（比如对交换性的考虑等）。此外，在涉及放大、缩小、旋转等情况时，还会用到线性代数的知识。而通过对渲染完毕的图像进行复用，可以制作出很酷的残影效果，但是要随心所欲地控制残影，就需要很多数学上的考量。不过大体上只要有高中程度的数学知识，就还是可以应对的。

然而随着时代的发展进步，可编程着色器（如顶点着色器、像素着色器等）隆重登场了。要掌握这些新工具，高中水平的数学知识就有些不够用了。比如通过像素着色器，可以将原图片的多个像素（像素点或纹理）混合为一个像素，即便这样复杂的处理仍然可以获得非常实时的处理速度。为此我们可以制作一个将图片模糊的画面切换效果，但为了实现“模糊”这个操作，就需要不少数学理论作为基础。虽然有时候比较简单的数学算法也能够实现一定的效果（比如圆锥滤波器），但为了实现一个更完美的模糊效果，就必须用到算法的曲线呈吊钟形的高斯滤波器。而为了理解这些算法，就需要各种数学知识（有些超过了高中数学的范畴）。比如使用着色器使波纹或者水面的倒影等扭曲的图像显现、消失时，一般可以用三角函数，但只用三角函数所能模拟的波纹效果并不真实。此时如果有一定的数学积累，就很容易想到用傅立叶逆变换，通过组合多个正弦波来制作出复杂的波形，此外根据具体情况，这时可能还需要用到类似“分形”等数学工具。上面提到的这些数学话题，基本上都超出了

高中数学的范围，需要大学程度（甚至更高）的数学知识。

时至今日，硬件性能又有了新的飞跃，现在的GPU 并行计算能力是以前根本不敢想象的。而这样的高速化的图像处理，极大地减少了算法的制约，即便是那些基于高等数学的复杂算法所驱动的画面特效，我们也可以实时对其加以运行。这对游戏开发者来说无疑是值得庆祝的，不过与此同时，开发者们也不能再以“这个特效硬件不能实时处理，没法加到游戏里”为由来回避难点了。在游戏开发中，与游戏开发人员一起工作的游戏策划人员，有时会因为缺少对技术实现难度、所需工时的正确认识，而提出一些不合理的要求，但现在游戏开发者就很难再用“硬件上无法实现”来拒绝这些不合理的要求了，所以从某些角度上来说，也许现在的游戏开发反而比以前更难了。回顾以前的游戏开发，当时主机（或平台）受到的硬件制约，在现在看来也是无法想象的，为了回避这些限制而产生的无数奇妙的技术，因不得不大量（甚至全部）采用汇编语言而导致的运行速度过慢、内存过小、软件工程、开发环境的不成熟（比如没有开发语言不支持面向对象）等，都让以前的游戏开发者无比痛苦。纵观全局，虽然时代进步了，但是游戏开发并没有变简单，也许只是难点从一个地方转移到了另一个地方吧。

笔者站在一名游戏学校的教育者的角度来看，虽然现在对人才的要求在不断提高，但是每年高中毕业进入游戏学校的学生的水平并没有随之增高，因此想要培养出能符合游戏公司要求的人才也越来越难。电视游戏产业的终端大多为日本制造，这也是日本为之骄傲的产业之一。电视游戏以外的其他产业，现在大都也面临着技术革新，对人才的要求也越来越高。在此笔者深切地希望现在的初中、高中，能够顺应时代的需要，对教

育方式进行更多的研究、实践，以更好的水平、更高的效率、更可行的教育方式培养出更多人才。

第 6 章　游戏开发的数学和物理学基础理论

6.1 比例、一次函数及直线方程

Key Word 比例系数、斜率、截距、参数方程 RANK/easy 前半部分 RANK/normal 后半部分

- 比例与一次函数

一个数可表示为另一个数乘以常数 a 时，称这两者“成比例关系”。此时 a 称为比例系数。比如一颗糖 10 日元，买糖花费的金额与糖的数量之间就成比例关系。令买到的糖的数量为 x，花费的金额为 y，可表示为

$$y = 10x$$

此时上面所说的比例系数 a 就等于 10。使用这个等式可以计算金额，比如让 x =5，即买 5 颗糖所需要的金额 y 为

$$y = 10 \cdot 5 = 50 \text{（日元）}$$

买 100 颗糖需要的金额 y 为

$$y = 10 \cdot 100 = 1000 \text{（日元）}$$

上面的等式还可以用其他形式表示。一颗糖等于 10 日元，那么糖∶金额 =1∶10（读作 1 比 10）。通过这种表示形式，两颗糖等于

20 日元，所以糖∶金额 =2∶2；同理，7 颗糖等于 70 日元，糖∶金额 =7∶70。这一组关系可总结为

糖∶金额 = 1∶10 = 2∶20 = 7∶70

请注意上式的 1∶10=2∶20 这一部分。等号两边内侧的两个数相乘有 10·2=20，外侧的两个数相乘有 1·20=20，结果相同（参考图 6-1-1）。

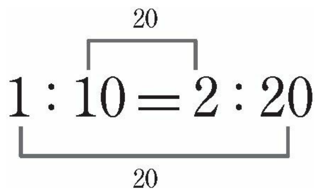


图 **6-1-1**　等式内侧两数相乘与外侧两数分别相乘

再对比一下 2 ∶ 20=7 ∶ 70 这部分。内侧的两个数相乘为 20·7=140，外侧的两个数相乘为 2·70=140，结果依然相同。那么我们可以得出结论：

当 a ∶ b =c ∶ d 时，bc =ad

这样的关系我们在 2.1 节、2.6 节中都有使用，有兴趣的话可以回顾一下。接下来我们就尝试用这样的关系来计算买糖花费的金额。糖 ∶ 金额 =1 ∶ 10，假设买 5 颗糖所需的金额为 y，有如下关系成立。

1∶10 = 5∶y

内侧的两个数相乘为 10·5=50，外侧的两个数相乘为 1·y =y，由此可以得到 y =50，即买 5 颗糖所需要的金额 y 为 50 日元。下面我们再尝试用糖 ∶ 金额 =2 ∶ 20 来计算

100 颗糖所花费的金额。

$2：20 = 100：y$

由于内侧的两个数相乘等于外侧的两个数相乘，因此有

$20 \cdot 100 = 2 \cdot y$
$2000 = 2y$
$\therefore y = \frac{2000}{2} = 1000$ （日元）

即买100 颗糖需要的金额为 1000 日元。

在上面这样的比例关系中，我们总是能找到一个常数 a 作为比例系数，且有

$y = ax$

的关系成立。上例中糖的数量为 x，金额为 y，比例系数 a =10。

在之前的章节中，出现过以每帧 3 个像素的速度移动的物体，此时假设帧数为 x，行进的距离为 y，则有

$y = 3x$

与买糖的例子一样，通过比例关系计算，可以很简单地计算出一定时间的移动距离。但是与买糖的例子不同的是，物体的运动是有初始位置的，而初始位置也需要体现在关系式中。比如物体开始在 10 的位置，并且以每帧 3 像素的速度行进，令经过的帧数为 x，位置为 y，则有

$y = 3x + 10$

这个关系可以归纳为，如果物体的初始位置

为 b ，并以每帧 a 像素的速度行进，则经过 x 帧后 所经过的距离 y 为

$$y = ax + b$$

这种形式的等式称为一次函数 。当 b =0 时，会得到之前的简单的比例关系式 y =ax ，因此这种简单的比例关系式也是一种一次函数。

- 一次函数的图形与参数

将一个一次函数表示为图形必定得到一条直线。在一次函数的图形中，a 称为斜率 ，b 称为截距 （参考图 6-1-2）。

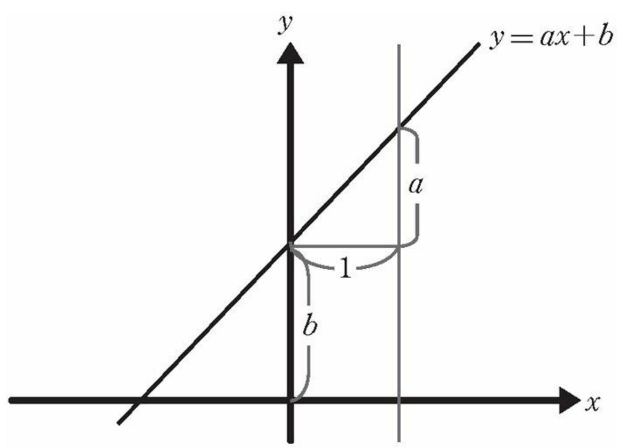


图 **6-1-2**　一次函数的图形

一次函数可以表示为一条直线，反过来直线也可以代表一个一次函数。但是平面上的直线并不是都可以用形如

$$y = ax + b$$

的一次函数来表示。比如与 y 轴平行的直线就不是一次函数。这样的直线在数学上可以被看作其斜率 a 为无穷大。此外，即使是没有完全与 y 轴平行，而只是相对 y 轴有微小

倾斜的直线，其截距 b 也可以被看作极大值，这种情况在计算机中是很难处理的（参考图 6-1-3）。

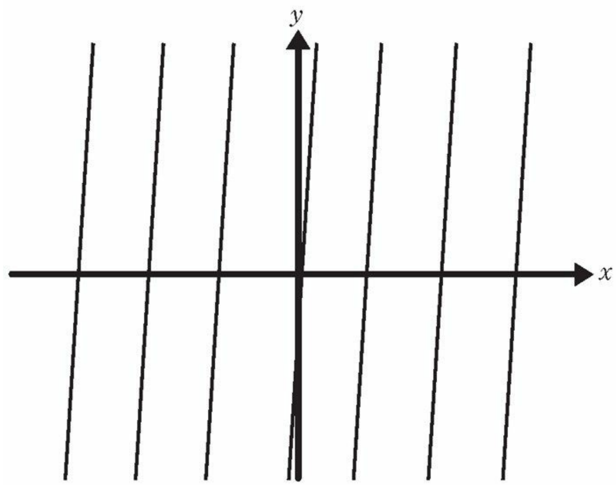


图 **6-1-3**　相对于 ***y*** 轴有微小倾斜的直线

于是，当计算机将直线作为图形处理时，都会引入参数 t ，并以下面这样的参数方程来表示原来的一次函数，这会让计算变得更加方便。

$$x = a_x t + b_x$$
$$y = a_y t + b_y$$

也就是说，参数方程将原来的一个一次函数，根据 x 坐标、y 坐标进行了拆分，平面的情况下就拆分为两个一次函数来共同表示一条直线（在 3.3 节、3.4 节，以及 5.2 节中都用过这种方法）。而上面的参数 t ，可以将其看作时刻，那么上面的等式就可以表示为“在时刻 0 时位于 (bx , by)，并以速度 (ax , ay) 运动的直线”。例如

$$x = 3t + 1$$
$$y = 4t + 2$$

这个参数方程，就表示在时刻 0 时位于位置

(1, 2)，并且以速度 (3, 4) 运动的一条直线。这种表示方法与 $y=ax+b$ 不同的是，即便是与 y 轴平行的直线，只要使 ax 为 0，也可以很自然地表示出来，非常方便。

在此基础上，如果对参数 t 的取值范围进行限制，就可以表示连接一点到另一点的线段，除此之外还有很多其他用途。像之前的

$$x = 3t + 1$$
$$y = 4t + 2$$

这条直线，将 t 的值限制在 $0 \le t \le 2$ 范围内，就可以很简单地表示连接点 (1, 2) 到点 (7, 10) 的线段（参考图 6-1-4）。

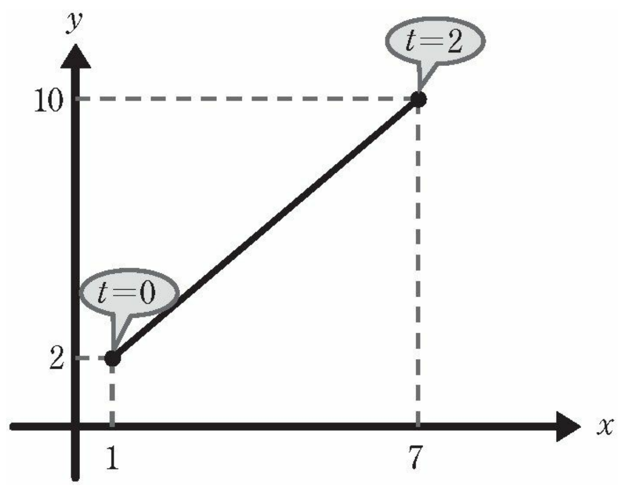


图 **6-1-4**　连接点 **(1, 2)** 到点 **(7, 10)** 的线段

像这样，在 2D（平面）中，使用两个一次函数组成一组参数方程来表示直线的方法，就可以很容易地表示各种各样的直线、线段，这在游戏中制作直线、线段时经常被使用。

6.2 算式展开与因式分解

Key Word 计算优先级、分配律

RANK/easy 前半部分 RANK/normal 后半部分

- 算式展开

  所谓算式展开，大致可以看作将一个先进行加法运算（或减法运算）后进行乘法运算的算式，变形为先进行乘法运算后进行加法运算的算式的操作。比如

  $$(1+2)(3+4)$$

  这个算式可以变形为

  $$1\times3+1\times4+2\times3+2\times4$$

  这种变形就是展开。由于括号内的加法（减法）运算需要比乘法运算优先执行，因此展开操作也可以被看作是消去括号的操作。而上面的算式，无论展开前后，都会得到 21 这个值。

  对于上面的例子，由于算式所包含的都是常数，进行算式展开并没有得到什么实际的好处，反而增加了计算量。但是

如果算式中含有变量，算式的展开就变得非常有意义了。比如程序中有

$(a_x t + b_x)^2 + (a_y t + b_y)^2 = r^2$ （t 为变量，a_x、b_x、a_y、b_y、r 为常数）

这样一个方程需要求解，也就是说，要根据这个等式将 t 通过 a_x、b_x、a_y、b_y、r 表示出来（在 3.4 节中遇到过类似问题），此时就必须将等式展开。对于上例这样的等式，程序中一般会将等式展开为$at^2 + bt + c = 0$这种形式，以便进一步通过二次方程的求根公式求解 t 。本小节将首先围 绕等式的展开进行一些介绍。

- 展开的基础：分配率

让我们从最简单的展开说起，来一起思考对于 $a(b+c)$ 这个算式，如何将其括号消去。按照原来的计算顺序，这个算式表示数 b 与数 c 相加，然后乘以数 c 。为了方便理解，可以将该等式想象成需要准备一些数字 a ，一共准备 $b+c$ 组。

举一个更具体的例子。比如一袋 5 颗糖，需要准备 3 袋 +7 袋，求一共有多少颗糖（即 a =5、b =3、c =7）。按算式原本的顺序计算，糖共有 3 袋 +7 袋 =10 袋，每袋分别装 5 颗糖，共计 5×10=50 颗。但是还有其他计算方法，比如 3 袋糖有 5×3=15 颗，7 袋则有 5×7=35 颗，总数是 15+35=50 颗，两次计算结果相同（参考图 6-2-1）。

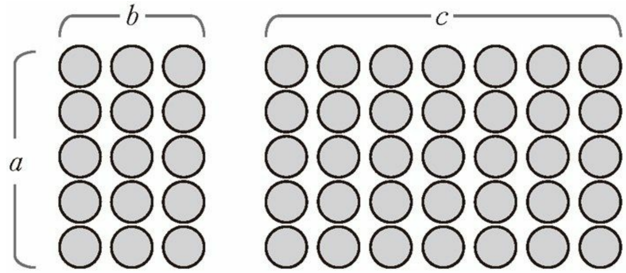


图 **6-2-1**　**5** 颗一袋的糖共 **3+7** 袋

将括号内的数相加之前，分别与括号外的数相乘再相加，与先相加再相乘的计算结果是相同的，因此等式 $a(b+c)$ 就可以表示为

$$a(b+c)=ab+ac$$

这就是展开的基础。请注意乘法的顺序对结果是没有影响的。

$$a(b+c)=(b+c)a=ab+ac$$

要乘的数放在后面也能得到相同结果。

如果加法部分的数量增加，计算方法也一样，比如 5 颗一袋的糖共 3 袋 +7 袋 +5 袋，可以套用同样的解法。

$$a(b+c+d)=ab+ac+ad$$

括号中的加法部分无论增加多少项，只要将每项的数字分配给 a 相乘就可以了，这在数学中称为分配律 。

- 更复杂的展开式以及 **2** 次方、**3** 次方的展开式

下面考虑更复杂的情况，$(a+b)(c+d)$ 这样的算式要如何展开呢？这个算式可以理解为，5 颗每袋的普通糖果 +2 颗

每袋的赠品糖果，一共准备 3 袋 +7 袋，求一共有多少颗糖果（参考图 6-2-2）。

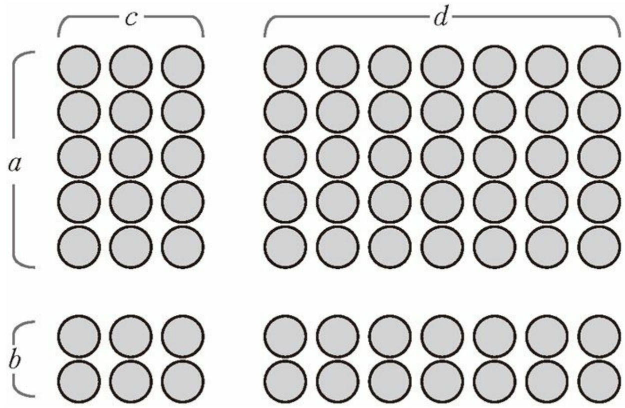


图 **6-2-2　5+2** 颗糖果共 **3+7** 袋

在日常生活中，可能一般会先计算 3 袋 +7 袋一共有多少袋。我们也可以采用同样的思路，对于算式 $(a+b)(c+d)$，将 $(c+d)$ 的部分看作一个数 e，先将 $(c+d)=e$ 置换进算式，

$$(a+b)(c+d)=(a+b)e$$

套用之前的分配律 $(b+c)a=ab+ac$ 可得

$$(a+b)e=ae+be$$

进一步将 $(c+d)=e$ 置换进去，把 e 替换为原来的 $(c+d)$。

$$ae+be=a(c+d)+b(c+d)$$

再次使用分配律继续展开可得

$$a(c+d)+b(c+d)=ac+ad+bc+bd$$

最终得到

$(a + b)(c + d) = ac + ad + bc + bd$

所以 5 颗每袋的普通糖果 +2 颗每袋的赠品糖果，一共准备 3 袋 +7 袋，总共需要普通装 5 个 × 3 袋 + 普通装 5 个 × 7 袋 + 赠品装 2 个 × 3 袋 + 赠品装 2 个 × 7 袋（参考图 6-2-2）。

另外，由上面的基本展开式还可以得到以下结论。

$(a + b)^2 = a^2 + 2ab + b^2$

有兴趣的读者可以根据 $(a + b)^2 = (a + b)(a + b)$，使用上面的展开式求解验证一下。使用这个结论我们就可以求解本小节一开始举例的方程

$(a_x t + b_x)^2 + (a_y t + b_y)^2 = r^2$（$t$ 为变量，a_x、b_x、a_y、b_y、r 为常数）

一起来试试吧。

$$(a_x t + b_x)^2 + (a_y t + b_y)^2 = r^2$$
$$a_x^2 t^2 + 2a_x b_x t + b_x^2 + a_y^2 t^2 + 2a_y b_y t + b_y^2 = r^2$$

展开到此为止，为了能使用二次方程的求根公式，需要合并次数相等的 t 项。

$$(a_x^2 + a_y^2)t^2 + 2(a_x b_x + a_y b_y)t + b_x^2 + b_y^2 - r^2 = 0$$

然后令 $a = a_x^2 + a_y^2$、$b = 2(a_x b_x + a_y b_y)$、$c = b_x^2 + b_y^2 - r^2$，使用二次方程的求根公式，就可以得到答案。

这种 2 次方的展开式，在游戏编程中经常被用于基于距离的碰撞检测等很多情

况，建议熟记。相比 2 次方，3 次方的展开式使用频度略低，展开过程如下。

$$\begin{aligned}(a+b)^3 &= (a+b)(a+b)^2 \\ &= (a+b)(a^2+2ab+b^2) \\ &= a^3+2a^2b+ab^2+a^2b+2ab^2+b^3 \\ &= a^3+3a^2b+3ab^2+b^3\end{aligned}$$

整理一下得到

$$(a+b)^3 = a^3+3a^2b+3ab^2+b^3$$

3 次方展开式虽然略复杂，不过还是有一些场景会用到。而如果要进一步推广到 $(a+b)^n$ 的展开式，则可以使用“二项式定理”，此处不再详细说明。

另外，为了方便地进行等式的变形，还有下面这样的展开式。

$$(x+a)(x-a) = x^2-a^2$$

感兴趣的读者可以将等式左边展开验证一下。这个展开式在二次方程的求根公式中经常用到，建议牢记。

- 算式展开的逆运算：因式分解

  与算式展开相反，将一个先进行乘法运算后进行加法运算的算式，变形为先进行加法运算（或减法运算）后进行乘法运算的算式，这样的操作叫作因式分解。比如将

  $$x^2+(\alpha+\beta)x+\alpha\beta$$

  这个等式变形为

  $$(x+\alpha)(x+\beta)$$

就完成了一次因式分解。至于这两个算式是否真的相等，可以将 $(x+\alpha)(x+\beta)$ 展开进行验证。一般来说，算式的展开只要按照规则一步步操作就可以，相对比较简单，因式分解则很容易遇到麻烦。但是因式分解仍然很重要，因为这是求解方程式所必需的。比如对

$$x^2 - 5x + 6 = 0$$

这个方程求解，先对左边进行因式分解，

$$(x - 3)(x - 2) = 0$$

就得到 $x=3$、$x=2$ 两个解。将其进一步推广，如果能对

$$ax^2 + bx + c = 0$$

这个等式的左边进行因式分解，就可以对二次方程的求根公式求解。这部分会在下一小节中详细介绍，请注意参考。

6.3　二次函数、二次方程与抛物线 · 圆

Key Word　完全平方、求根公式、圆锥曲线

RANK/normal 后半部分

- 二次函数

二次函数是指形如 $y=ax^2+bx+c$ 的函数，其特征是包含 x^2 这样一个平方项。最简单的二次函数的形式为 $y=ax^2$，其图形如图 6-3-1 所示。

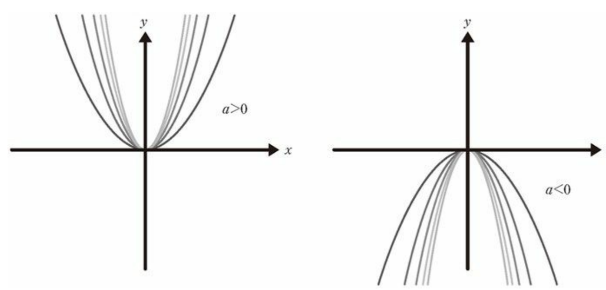


图 **6-3-1**　最简单的二次函数的图形（抛物线）

这样的曲线称为抛物线，顾名思义，向空中抛出一个物体的运动轨迹会形成这样的曲线（其实由于空

气阻力等原因，物体实际运动的轨迹并不是严格的抛物线）。由于 x^2 是一个数自己与自己相乘，当 x 为正时显然 x^2 也为正；当 x 为负时，负数 × 负数仍然为正，因此 $y = ax^2$ 这个函数在 a >0 时，无论 x 为任意实数，y 都为正数或 0；而当 a ＜0 时，无论 x 为任意实数，y 都为负数或 0。也就是说，$y = ax^2$ 这个函数只可能取正数与 0（图 6-3-1 左），或负数与 0（图 6-3-1 右）。

- 二次方程的完全平方

在 $y = ax^2$ 这样一个含有 x 平方项的函数的基础上，加上 x 的一次方项以及常数 c，就得到 了二次函数的一般形式 $y = ax^2 + bx + c$。这个等式所表示的图形，等同于将 $y = ax^2$ 的图形向 x 方向及 y 方向平移。也就是说，对 ax^2 加上一次方程式 $bx + c$，只会改变其位置而不会对图形本身的形状产生影响。是不是有些不相信呢？接下来就让我们对 $y = ax^2 + bx + c$ 做一些变形来证明这个结论。

首先考虑将等式先变形为 $y = a\square^2 + \triangle$ 的形式。如果能实现变形，变形后的等式就能表示抛物线向 y 方向平移 △。为此我们希望将 x 的一次项部分 bx 消去，于是就需要设法将 bx 并入 x 的二次项中。我们先将 bx 部分强制乘以一个系数 a，得到如下等式。

$$y=a\left(x^2+\frac{b}{a}x\right)+c$$

然后设法将这个等式括号内的部分变为平方的形式。请大家回想一下前一小节中讲过的 平方的展开式 $(x+\alpha)^2=x^2+2\alpha x+\alpha^2$，是不是有一些启发呢？由于展开式中 x 的一次项系数为 2α，而此处括号内的一次项系数为 $\frac{b}{a}$，那么令 $2\alpha=\frac{b}{a}$，即 $\alpha=\frac{b}{2a}$，这样我们就向平方形式迈进了一步。但是还缺少展开式的 α^2 部分，即缺少 $\left(\frac{b}{2a}\right)^2$，我们可以使用一个小技巧，先加上 $\left(\frac{b}{2a}\right)^2$，再减去同样的项，最后得到

$$y=a\left\{x^2+2\cdot\frac{b}{2a}x+\left(\frac{b}{2a}\right)^2-\left(\frac{b}{2a}\right)^2\right\}+c$$

这样一来，花括号中的前3项，正好满足 $x^2+2\alpha x+\alpha^2$ 的形式，将其整理为 $(x+\alpha)^2$ 的形式，有

$$y=a\left\{\left(x+\frac{b}{2a}\right)^2-\left(\frac{b}{2a}\right)^2\right\}+c$$

花括号内多余的 $-\left(\frac{b}{2a}\right)^2$ 项，是不包含 x 的常数，可以移到花括号外。

$$y=a\left(x+\frac{b}{2a}\right)^2-a\cdot\frac{b^2}{4a^2}+c$$

将移出去的常数，与花括号外原有的常数 c 合并并进行通分，有

$$y=a\left(x+\frac{b}{2a}\right)^2-\frac{b^2-4ac}{4a}$$

再令 $X=x+\frac{b}{2a}$，有

$$y=aX^2-\frac{b^2-4ac}{4a}$$

由于 $x=X-\frac{b}{2a}$，因此 $y=ax^2+bx+c$ 的图形，就等于将 $y=ax^2$ 的图形平移 $\left(-\frac{b}{2a},-\frac{b^2-4ac}{4a}\right)$（参考图6-3-2）。将二次函数变形为这种形式的等式，叫作完全平方 。

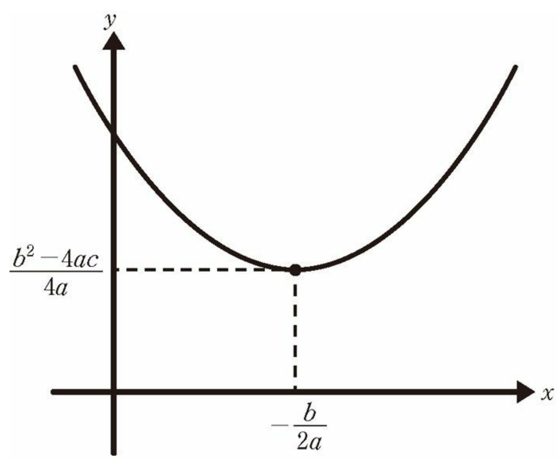


图 **6-3-2**　平行移动的图形

- 计算二次方程的求根公式

上面通过完全平方得到了二次函数，令 y =0，就可以对二次方程

$$ax^2 + bx + c = 0$$

求解。如果二次方程比较简单，可以直接通过因式分解手工算出结果，而当二次方程比较复杂时，往往我们会用到二次方程的求根公式，通过求根公式可以对任意 a 、b 、c 值的 x 求解，特别是在用计算机来求解二次方程时，这种方法非常方便。下面就让我们实际尝试一下，对于方程

$$ax^2 + bx + c = 0$$

从刚才的结果可知

$$a\left(x+\frac{b}{2a}\right)^2-\frac{b^2-4ac}{4a}=0$$

两边同除以 a ，有

$$\left(x+\frac{b}{2a}\right)^2-\frac{b^2-4ac}{4a}=0$$

对左边进行因式分解，得到

$$\left\{\left(x+\frac{b}{2a}\right)+\sqrt{\frac{b^2-4ac}{4a^2}}\right\}\left\{\left(x+\frac{b}{2a}\right)-\sqrt{\frac{b^2-4ac}{4a^2}}\right\}=0$$

进而有

$$\begin{aligned}\left(x+\frac{b}{2a}\right)&=\pm\sqrt{\frac{b^2-4ac}{4a^2}}\\&=\pm\frac{\sqrt{b^2-4ac}}{2a}\end{aligned}$$

将 $\frac{b}{2a}$ 移项到右边有

$$x = -\frac{b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

这称为二次方程的求根公式 。这个等式在游戏编程中请务必牢记。

而如果二次方程是

$ax^2 + 2bx + c = 0$

这样的形式，即 x 的一次项乘以了系数 2，则解会稍微简单一些，实际求解过程如下。

$$\begin{aligned}
ax^2 + 2bx + c &= 0 \\
a\left(x^2 + 2\frac{b}{a}x\right) + c &= 0 \\
a\left\{x^2 + 2\frac{b}{a}x + \left(\frac{b}{a}\right)^2 - \left(\frac{b}{a}\right)^2\right\} + c &= 0 \\
a\left\{\left(x + \frac{b}{a}\right)^2 - \left(\frac{b}{a}\right)^2\right\} + c &= 0 \\
a\left(x + \frac{b}{a}\right)^2 - \frac{b^2}{a} - +c &= 0 \\
a\left(x + \frac{b}{a}\right)^2 - \frac{b^2 - ac}{a} &= 0 \\
\left(x + \frac{b}{a}\right)^2 - \frac{b^2 - ac}{a^2} &= 0 \\
\left\{\left(x + \frac{b}{a}\right) + \sqrt{\frac{b^2 - 4ac}{a^2}}\right\}\left\{\left(x + \frac{b}{a}\right) - \sqrt{\frac{b^2 - ac}{a^2}}\right\} &= 0 \\
\left(x + \frac{b}{a}\right) &= \pm\frac{\sqrt{b^2 - ac}}{a} \\
x &= -\frac{b}{a} \pm \frac{\sqrt{b^2 - ac}}{a} \\
x &= \frac{-b \pm \sqrt{b^2 - ac}}{a}
\end{aligned}$$

上面的解在游戏开发中也非常实用，希望读者可以记下来（在 3.4 节中使用过）。

- 圆的方程

上面的二次函数中只有变量 x 包含平方项，如果不单是 x，变量 y 也包含平方项，那么应该如何处理呢？这种情况下，我们一般会将函数对应圆锥曲线 。根据各项系数值的不同，圆锥曲线可能呈现双曲线、椭圆等形状。而圆锥曲线中最重要的是圆的方程。

圆的方程一般形如

$$x^2 + y^2 + ax + by + c = 0$$

不过这样的等式是否真的能表示一个圆，我们很难直观地看出来，而圆的一些特征值，比如圆的半径及中心位置等，也无法直接得到。因此需要将上面的等式进行变形。

$$\begin{aligned} x^2 + y^2 + ax + by + c &= 0 \\ x^2 + ax + \left(\frac{a}{2}\right)^2 - \left(\frac{a}{2}\right)^2 + y^2 + by + \left(\frac{b}{2}\right)^2 - \left(\frac{b}{2}\right)^2 + c &= 0 \\ \left(x + \frac{a}{2}\right)^2 - \left(\frac{a}{2}\right)^2 + \left(y + \frac{b}{2}\right)^2 - \left(\frac{b}{2}\right)^2 + c &= 0 \\ \left(x + \frac{a}{2}\right)^2 + \left(y + \frac{b}{2}\right)^2 &= \left(\frac{a}{2}\right)^2 + \left(\frac{b}{2}\right)^2 - c \end{aligned}$$

上面等式中包含的 $\frac{a}{2}$、$\frac{b}{2}$、$\left(\frac{a}{2}\right)^2 + \left(\frac{b}{2}\right)^2 - c$ 等虽然形式看起来有点复杂，其实全部都只是常数。为了理解方便，可以将其替换为其他常数整理一下。具体来说，将 $\frac{a}{2}$ 替换为 $-x_0$，将 $\frac{b}{2}$ 替换为 $-y_0$，并将等式右边的 $\left(\frac{a}{2}\right)^2 + \left(\frac{b}{2}\right)^2 - c$ 替换为 r^2，

$$\left(x+\frac{a}{2}\right)^2+\left(y+\frac{b}{2}\right)^2=\left(\frac{a}{2}\right)^2+\left(\frac{b}{2}\right)^2-c$$

这个等式就变为了

$$(x-x_0)^2+(y-y_0)^2=r^2$$

这个等式正好与勾股定理完全一致，变量 x 、y 表示从坐标 $(x_0, y_0$ 出发、距离为 r 的一点。而这样的点的集合就是一个圆，因此方程

$$(x-x_0)^2+(y-y_0)^2=r^2$$

表示中心坐标为 (x_0, y_0)、半径为的圆。这个等式在游戏编程中非常重要，建议牢记（在 3.4 节 中有使用）。

6.4　三角函数

Key Word　直角三角形、单位园、弧度、相位

RANK/normal

- 三角函数

  所谓三角函数，顾名思义，就是关于三角形边长与角度的函数。三角函数有很多种类，在游戏编程中特别重要的是cos、sin、tan 三个。在这 3 个函数的原始的定义中，对于直角三角形的一个锐角 θ，有如下关系成立（参考图 6-4-1）。

$$\cos\theta = \frac{a}{c}$$

$$\sin\theta = \frac{b}{c}$$

$$\tan\theta = \frac{b}{a}$$

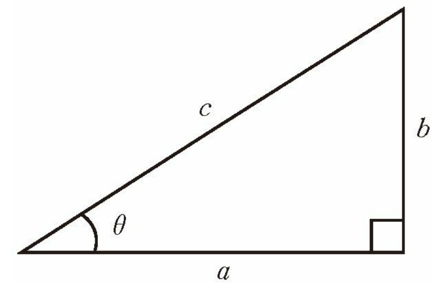


图 **6-4-1**　直角三角形的三边与角 ***θ***

这些函数之所以在游戏开发中很重要，主要是因为当坐标上有一点 $P(x, y)$（$x \geq 0$ 且 $y \geq 0$）时，如果令点 P 与原点 O 连接成的线段长为 r，线段 OP 与 x 轴的夹角为 θ，则可以根据三角函数很容易地得到以下关系式。

$$\cos\theta = \frac{x}{r}$$
$$\sin\theta = \frac{y}{r}$$
$$\tan\theta = \frac{y}{x}$$

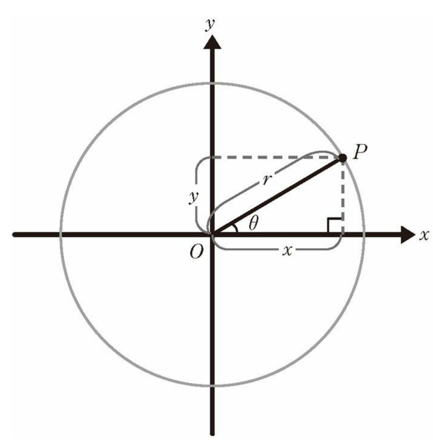


图 **6-4-2**　将坐标上一点与原点连接构成的三角函数关系

对 cos 与 sin 的等式进行变形，即可得到

$$x = r \cdot \cos\theta$$
$$y = r \cdot \sin\theta$$

这是非常有用的关系式。利用这个等式，将 r 固定，让 θ 进行变化，就可以产生半径为 r 的旋转运动（参考图 6-4-3 左，在 1.6 节中有使用）。而将角度 θ 固定，让 r 进行变化，则可以让物体沿角度 θ 的方向移动（参考图 6-4-3 右，在 1.3 节中有使用）。

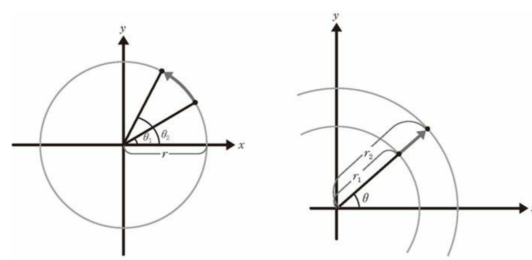


图 **6-4-3**　使用三角函数移动坐标

tan 函数在游戏开发中直接使用的机会要少一些，不过 tan 的反函数 $\tan^{-1}$ 却经常被用到。

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

据此可以方便地根据坐标 (x , y) 计算夹角。特别是当 x 的绝对值极小时，不用考虑异常处理也能得到正确象限的角度，所以希望大家能更好地利

用 C 语言中的 atan2 函数。

- 不限定角度范围的三角函数定义

上面讨论了三角函数（atan2 函数除外）在 $x \geq 0$ 且 $y \geq 0$ 时的情况，即将坐标系只限定在了第一象限内。这是因为三角函数 cos、sin、tan 是根据直角三角形定义的，所以角度 θ 必须小于 90 度。而为了使不限定角度 i 的范围时上面的结论仍然成立，我们需要引入“单位圆”的概念来重新定义三角函数。假设有以原点 O 为中心、半径为 1 的圆（即单位圆），单位圆上有一点 $P(x, y)$，则有如下定义（参考图 6-4-4）。

$$\cos\theta = x$$
$$\sin\theta = y$$
$$\tan\theta = \frac{y}{x}$$

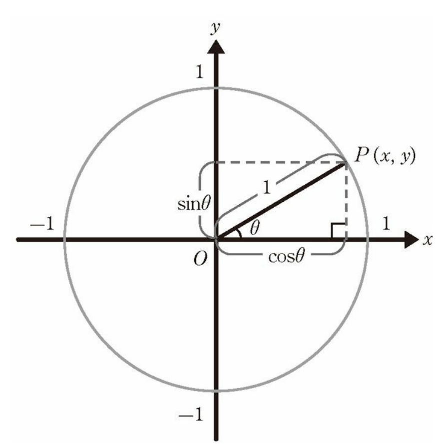


图 **6-4-4**　基于单位圆的三角函数

这样一来，三角函数的定义就不再限定角度，可以应用于任意象限的坐标，变得更加方便。计算机中的三角函数cos、sin、tan等，都是基于单位圆进行定义的，因此无需担心角度太大无法使用的问题。

下面我们来考虑一下角 θ 的单位。在日常生活中，角的单位一般都使用角度（一周为360度），而在高等数学及计算机中，一般使用弧度 作为三角函数的单位。比如在单位圆中，角度正好等于单位圆上一段弧的长度（参考图6-4-5），单位圆一周的周长为 2π ，所以一周的角（360度）以弧度为单位就为 2π ，同理，30度为 $\frac{\pi}{6}$ 、60度为 $\frac{\pi}{3}$ 、90度为 $\frac{\pi}{2}$ 、180度正好等于 π 。

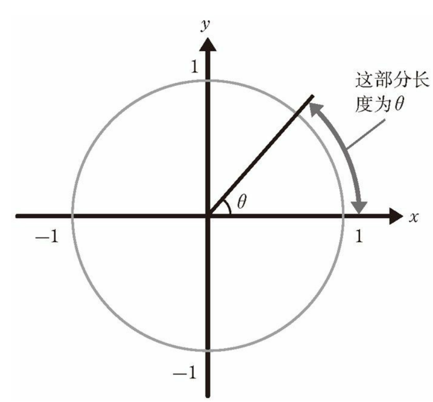


图 **6-4-5**　弧度的定义

使用弧度为单位，可以很容易地对三角函数进行微分积分，非常实用。事实上在游戏开发

中，几乎没有角度的用武之地，绝大多数情况下都会以弧度为单位。

此外一个角的角度还可能被定义为负（参考图 6-4-6）。在以 *y* 轴上方为正的坐标系中，角度的正方向（即角度增加的方向）被规定为逆时针方向，角度的负方向与此相反，为顺时针方向（但是在计算机中 2D 坐标一般以 *y* 轴下方为正，因此方向正好是颠倒的）。

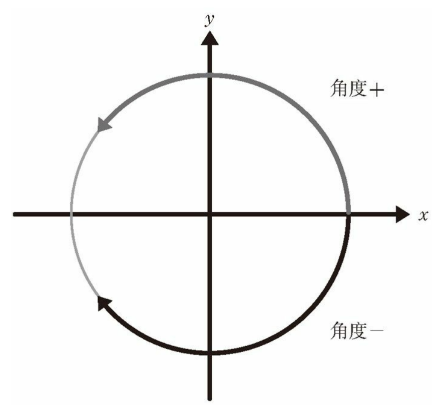


图 **6-4-6**　负角

既然还有负角度的定义，那么所有值，即在全体实数范围内就都可以定义三角函数（但是 tan 要除外，tan 在一部分值范围内是无法定义的）。这样一来，本来仅在直角三角形，即 0 度～90 度范围内才有定义的三角函数，一下子有了极大的扩展空间。在实数范围内定义的三角函数 cos、sin、tan 的图形如图 6-4-7 所示。可以看

到 cos 和 sin 的图形完全相同，只是在 x 轴方向的位置有所差异。

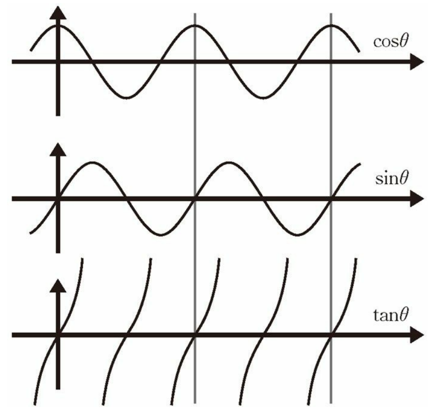


图 **6-4-7**　在所有实数范围内定义的三角函数的图形

- 加法定理与积化和差

对于定义域扩展到全体实数范围的三角函数来说，有各种各样的公式成立。这些公式不但数量繁多，其中有一些还非常复杂，这也是大家对三角函数敬而远之的原因。但是接下来我们还是要从这些公式中挑选一个进行说明，那就是对于游戏开发极为重要的 cos、sin 的加法定理 。所谓加法定理，是有关 $\cos(\alpha+\beta)$ 以及 $\sin(\alpha+\beta)$ 的公式，为了理解公式的意义，大家可以将其想象成让位于单位圆上角度 α 位置的一点，再旋转角度 β（参考图 6-4-8）。由于是将一点旋转某个角度，因此在游戏开发中使用旋转矩阵（参考 6.5 节）可能会比较容易理解。

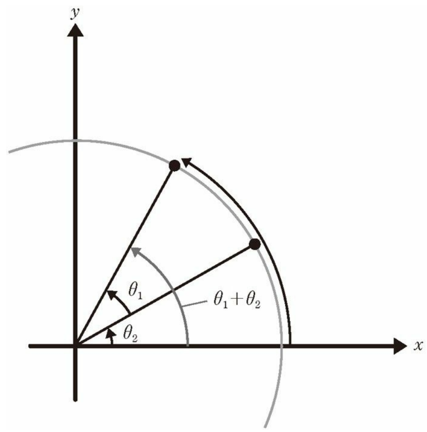


图 **6-4-8**　表示加法定理的图

具体来说，点 (cos(*α* +*β*), sin(*α* +*β*)) 可以由点 (cos *α* , sin *α*) 旋转角度 *β* 得到，使用旋转矩阵则有

$$\begin{pmatrix}\cos(\alpha+\beta)\\ \sin(\alpha+\beta)\end{pmatrix}=\begin{pmatrix}\cos\beta & -\sin\beta\\ \sin\beta & \cos\beta\end{pmatrix}\begin{pmatrix}\cos\alpha\\ \sin\alpha\end{pmatrix}$$

由这个等式的 *x* 部分可知

$\cos(\alpha+\beta)=\cos\alpha\cos\beta-\sin\alpha\sin\beta$

由等式的 *y* 部分可知

$\sin(\alpha+\beta)=\cos\alpha\sin\beta+\sin\alpha\cos\beta$

上面这两个等式经常被用于包含三角函数的等式变形中，所以记下来会比较方便。在游戏开发中，将 $\frac{\pi}{2}$ 代入 *β* 往往会得到一些方便的结论，实际来试一下。

$\cos(\alpha+\beta)=\cos\alpha\cos\beta-\sin\alpha$

$\sin\beta$

代入 $\beta = \frac{\pi}{2}$ 得

$$\begin{aligned}\cos\left(a+\frac{\pi}{2}\right) &= \cos\alpha\cos\left(\frac{\pi}{2}\right) - \sin\alpha\sin\left(\frac{\pi}{2}\right) \\ &= \cos\alpha\cdot 0 - \sin\alpha\cdot 1 \\ &= -\sin\alpha\end{aligned}$$

随着角度增加 $\frac{\pi}{2}$，cos 变成了 -sin。接下来对 sin 进行同样的处理。

$\sin(\alpha + \beta) = \cos\alpha \sin\beta + \sin\alpha \cos\beta$

代入 $\beta = \frac{\pi}{2}$ 可得

$$\begin{aligned}\sin\left(a+\frac{\pi}{2}\right) &= \cos\alpha\sin\frac{\pi}{2} + \sin\alpha\cos\frac{\pi}{2} \\ &= \cos\alpha\cdot 1 - \sin\alpha\cdot 0 \\ &= \cos\alpha\end{aligned}$$

可以看到随着角度增加 $\frac{\pi}{2}$，sin 变为了 cos。

观察三角函数的图形（图 6-4-7）可以看出，cos 与 sin 其实只有角度位置（称为相位 ）的差异， 上面的等式正好说明了这一点。而如果令上面等式中的 $\beta = -\frac{\pi}{2}$，则有

$$\cos\left(a-\frac{\pi}{2}\right)=\sin\alpha$$
$$\sin\left(a-\frac{\pi}{2}\right)=-\cos\alpha$$

像这样将角度旋转 $\frac{\pi}{2}$ 或 $-\frac{\pi}{2}$，让三角函数之间进行转换的方法，在求向量的垂直向量时非常方便，希望读者朋友最好能记下来。

而对于等式

sin(α + β) = cosα sinβ + sinα cosβ

如果将 β 替换为 -β， 由于正弦函数是奇函数，有sin (-β)=-sin β； 余弦函数为偶函数，有cos (-β)= cos β， 因此

sin(α - β) = -cosα sinβ + sinα cosβ

然后用之前的两个等式相减可得

sin(α + β) - sin(α - β) = 2cosα sinβ

如果进一步令α +β = A 、α -β = B， 根据 $\alpha=\frac{A+B}{2}$ 、$\beta=\frac{A-B}{2}$ 可得

$$\sin A-\sin B=2\cos\frac{A+B}{2}\sin\frac{A-B}{2}$$

即可以将三角函数的减法转化为乘法。这样的公式叫作积化和差公式，根据加减的三角函数的不同，积化和差公式也有很多变形，这里不再详述。积化和差公式在游戏开发中用到的地方不多，在涉及微分等理论时可能会派上用场。其实上面列举的将正弦函数减法运算转换为乘法运算的公式，在三角函数的微分公式中已经有用到，建议记下来。

6.5 向量与矩阵

Key Word 长度、方向、一次变换、逆变换

RANK/normal

- 向量

  所谓向量，就是有长度及方向的量，一般由多个标量（scalar，即单纯的数字）组合而成。比如由两个标量组合而成的二维向量，可以表示二维空间（平面）中有长度及方向的量。由三个标量组成的三维向量，可以表示三维空间中具有长度及方向的量。同理也可以扩展到四维向量，在游戏中一般被作为四元数（quaternion）使用。

  请注意向量与有向线段是不同的。有向线段有起点和终点，而向量则与起点无关，平行移动一个向量得到的还是原来的向量，因为表示一个二维空间的

有向线段时，需要起点及终点的 xy 坐标，总计 4 个标量，而表示二维空间的向量时则只需要两个标量就足够了。

向量不仅能以位置向量的形式表示空间上的位置，此外如速度、加速度等物理量，或者直线、圆等图形，甚至旋转等坐标变换等都可以用向量来表示，在游戏开发（特别是 3D 游戏）中，向量是非常重要的工具。

向量在数学上的表现形式有很多种，首先当变量名写作粗体的 $\boldsymbol{a}$ 时，这个变量就可以表示一个向量。向量也可以用 a 这种在变量名上加上箭头的形式来表示，不过这种形式在高等数学中并不常用。本书中统一以粗体字 $\boldsymbol{a}$ 形式的变量表示向量。

向量还可以表示为若干个标量的组合，有一些书中倾向于用这种形式。比如表示一个三维向量时，可以用三维空间的坐标（x , y , z ）表示（x , y , z 为标量），这称为行向量，或者写作 $\begin{pmatrix}x\\y\\z\end{pmatrix}$，称为列向

量 。

向量有以下属性。

- 绝对值

  向量的绝对值的定义如下。

  二维向量 $\boldsymbol{a}=\begin{pmatrix}a_x\\a_y\end{pmatrix}$ 的绝对值为

  $$|\boldsymbol{a}|=\sqrt{a_x^2+a_y^2}$$

  三维向量 $\boldsymbol{a}=\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}$ 的绝对值为

  $$|\boldsymbol{a}|=\sqrt{a_x^2+a_y^2+a_z^2}$$

  这是由勾股定理的公式得来的，向量的长就是其绝对值。对于普通的数字标量来说，直线上一点距离原点的长度等于其绝对值，这也适用于向量，因此向量的长等于其绝对值。

- 单位向量

  单位向量 就是绝对值（长度）为 1 的向量。

对于二维向量 $\boldsymbol{a}=\begin{pmatrix}a_x\\a_y\end{pmatrix}$ 来说，当

$$|\boldsymbol{a}|=\sqrt{a_x^2+a_y^2}=1$$

时为单位向量。对于三维向量 $\boldsymbol{a}=\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}$ 来说，当

$$|\boldsymbol{a}|=\sqrt{a_x^2+a_y^2+a_z^2}=1$$

时为单位向量。一般表示单位向量时，可以在向量的变量名上加上t符号，写作 $\hat{\boldsymbol{a}}$ 、$\hat{\boldsymbol{b}}$ 。单位向量在游戏开发中一般被作为基准向量，应用十分广泛，请读者牢记。

- 向量的运算

  与使用普通数字的标量相同，向量之间也有各种运算，比如下面这些。

  - 加法

    两个二维向量 $\boldsymbol{a}=\begin{pmatrix}a_x\\a_y\end{pmatrix}$ 与 $\boldsymbol{b}=\begin{pmatrix}b_x\\b_y\end{pmatrix}$ 的加法运算为

    $$\boldsymbol{a}+\boldsymbol{b}=\begin{pmatrix}a_x\\a_y\end{pmatrix}+\begin{pmatrix}b_x\\b_y\end{pmatrix}=\begin{pmatrix}a_x+b_x\\a_y+b_y\end{pmatrix}$$

    两个三维向量

$\boldsymbol{a}=\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}$ 与 $\boldsymbol{b}=\begin{pmatrix}b_x\\b_y\\b_z\end{pmatrix}$ 的加法运算为

$$\boldsymbol{a}+\boldsymbol{b}=\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}+\begin{pmatrix}b_x\\b_y\\b_z\end{pmatrix}=\begin{pmatrix}a_x+b_x\\a_y+b_y\\a_z+b_z\end{pmatrix}$$

即相同分量之间分别相加，就是向量间的加法运算。不要问这是为什么，因为就是这样定义的。如下图所示，向量的加法运算，等于将各向量的起点与终点相连（参考图 6-5-1）。

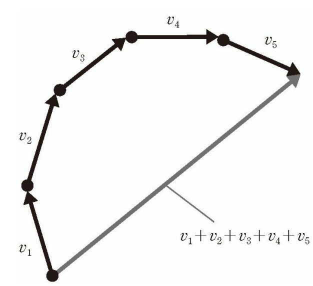


图 **6-5-1**　向量的加法运算

- 减法

两个二维向量 $\boldsymbol{a}=\begin{pmatrix}a_x\\a_y\end{pmatrix}$ 与 $\boldsymbol{b}=\begin{pmatrix}b_x\\b_y\end{pmatrix}$ 的减法运算为

$$\boldsymbol{a}-\boldsymbol{b}=\begin{pmatrix}a_x\\a_y\end{pmatrix}-\begin{pmatrix}b_x\\b_y\end{pmatrix}=\begin{pmatrix}a_x-b_x\\a_y-b_y\end{pmatrix}$$

两个三维向量 $\boldsymbol{a}=\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}$ 与 $\boldsymbol{b}=\begin{pmatrix}b_x\\b_y\\b_z\end{pmatrix}$ 的减法运算为

$$\boldsymbol{a}-\boldsymbol{b}=\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}-\begin{pmatrix}b_x\\b_y\\b_z\end{pmatrix}=\begin{pmatrix}a_x-b_x\\a_y-b_y\\a_z-b_z\end{pmatrix}$$

即相同分量之间分别做减法，就是向量间的减法运算。从图形来看，向量 $\boldsymbol{a}$ - $\boldsymbol{b}$，等于一条从向量 $\boldsymbol{b}$ 的前端指向向量 $\boldsymbol{a}$ 的前端的向量（参考图 6-5-2）。

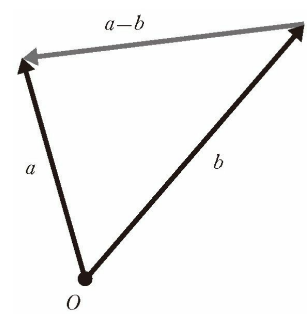


图 **6-5-2**　向量的减法运算

假设 $\boldsymbol{a}$ - $\boldsymbol{b}$ = $\boldsymbol{c}$，可以得到

$\boldsymbol{a}$ = $\boldsymbol{b}$ + $\boldsymbol{c}$

可见构成向量的等式中也可以进行普通的移项，这对应到图形上也不难理解。因此对于向量的加法、减法运算，完全可以像普通的等式那样进行等式的变形。

- 常数（标量）倍

  对二维向量 $\boldsymbol{a} = \begin{pmatrix} a_x \\ a_y \end{pmatrix}$ 乘以一个常数 α（标量），有

  $$\alpha \boldsymbol{a} = \alpha \begin{pmatrix} a_x \\ a_y \end{pmatrix} = \begin{pmatrix} \alpha a_x \\ \alpha a_y \end{pmatrix}$$

  同理，对三维向量 $\boldsymbol{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}$（$a_x$、$a_y$、$a_z$ 均为标量）乘以常数 α（标量），有

  $$\alpha \boldsymbol{a} = \alpha \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} = \begin{pmatrix} \alpha a_x \\ \alpha a_y \\ \alpha a_z \end{pmatrix}$$

  也就是说，向量的常数倍等于对向量的所有分量乘以同一常数。这也是向量常数倍的定义，所以没有办法再进一步解释。

  将向量的常数倍与向

量的加减法结合起来，就有了向量的结合律，如下所示。

$\boldsymbol{p} = \alpha \boldsymbol{a} + \beta \boldsymbol{b}$

这样得到的向量 $\boldsymbol{p}$ 会呈现出很多有趣的性质，例如，根据常数 α 与 β 的值的不同，其取值范围也有所不同，这会在后文详细介绍。

- 内积

何谓向量的内积？例如，对于两个二维向量 $\boldsymbol{a} = \begin{pmatrix} a_x \\ a_y \end{pmatrix}$ 与 $\boldsymbol{b} = \begin{pmatrix} b_x \\ b_y \end{pmatrix}$ 来说，它们的内积为

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_x b_x + a_y b_y$$

光看上面的等式可能无法明白内积到底代表什么，其实我们可以进一步得到以下关系。

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_x b_x + a_y b_y = |\boldsymbol{a}||\boldsymbol{b}| \cos \theta$$

其中 θ 是 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 的夹角。当 θ 为直角时 $\cos\theta$ 为 0，这可以被用于检测两向量是否

正交，以及求 $\cos\theta$ 的值等，在游戏开发中非常方便。

当向量扩展到三维时，两个三维向量 $\boldsymbol{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}$ 与 $\boldsymbol{b} = \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$ 的内积为

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_x b_x + a_y b_y + a_z b_z = |\boldsymbol{a}||\boldsymbol{b}| \cos\theta$$

- 外积

  所谓向量的外积，是指对于两个三维向量 $\boldsymbol{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}$ 与 $\boldsymbol{b} = \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$（二维向量中不存在外积的定义），有以下关系成立。

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

  外积与内积不同，其结果仍为一个向量。不过上面这个等式与内积一样，不容易直接看到其作用，因此

我们进一步将其转化为以下关系。

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix} = (|\boldsymbol{a}||\boldsymbol{b}|\sin\theta)\hat{\boldsymbol{n}}$$

其中 $\hat{\boldsymbol{n}}$ 是同时与向量 $\boldsymbol{a}$ 及向量 $\boldsymbol{b}$ 垂直的单位向量，因此外积得到的是与两个向量同时正交的向量。这可以被用于计算与某一物体正交的向量，即物体的法线向量，也可以被用于判定两个向量的位置关系，因为 $\sin\theta$ 的符号（为 + 还是为 -）与 θ 的符号一致，所以将外积的 z 分量 $a_x b_y - a_y b_x$ 单独取出来，就可以在平面内判定两个向量的位置关系（对于向量 $\boldsymbol{a}$ 来说向量 $\boldsymbol{b}$ 在其左侧还是右侧，这在 4.4 节中使用过）。

- 结合律

  向量的结合律可以表示为以下形式。

  $\boldsymbol{p} = \alpha\,\boldsymbol{a} + \beta\,\boldsymbol{b}$

  其中 α 与 β 为常数。通过更改 α 与 β，就可以由原来的向量 $\boldsymbol{a}$

与向量 $\boldsymbol{b}$，得到各种各样的新向量。比如当 $\alpha+\beta=1$ 时，令 $\beta=t$，由 $\alpha+\beta=1$ 可得 $\alpha=1-t$，则有

$$\boldsymbol{p}=(1-t)\,\boldsymbol{a}+t\,\boldsymbol{b}$$

再将其变形，有

$$\boldsymbol{p}=\boldsymbol{a}+t\,(\boldsymbol{b}-\boldsymbol{a})$$

向量 $\boldsymbol{p}$ 是以向量 $\boldsymbol{a}$ 的前端为起点，向向量 $(\boldsymbol{b}-\boldsymbol{a})$ 方向（当 $t<0$ 时为反方向）延伸的向量。此时的向量 $\boldsymbol{p}$，当 $t=0$ 时与向量 $\boldsymbol{a}$ 一致，当 $t=1$ 时与向量 $\boldsymbol{b}$ 一致，而当 t 在 0 到 1 的区间内变化时，其变化轨迹就构成了一条连接向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 的向量（参考图 6-5-3）。

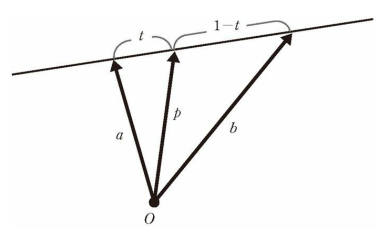


图 **6-5-3**　向量的内分

通过这种方式作出的向量 $\boldsymbol{p}$，是将向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 按 t ∶ $1-t$

内分。而等式

$$\boldsymbol{p} = (1 - t)\,\boldsymbol{a} + t\,\boldsymbol{b} \quad (0 \le t \le 1)$$

就称为向量的内分公式 。在游戏开发中，可以将这个公式中的 t 解释为时间，只要知道位置 $\boldsymbol{a}$ 与位置 $\boldsymbol{b}$ ，让物体在两位置间运动，就可以计算出位置 $\boldsymbol{a}$ 与位置 $\boldsymbol{b}$ 之间所有点的位置，这称为线性补间 ，在游戏开发中经常使用。

而对于向量的结合律

$$\boldsymbol{p} = \alpha\,\boldsymbol{a} + \beta\,\boldsymbol{b}$$

根据 α 与 β 的符号，还可以判定向量 $\boldsymbol{p}$ 与向量 $\boldsymbol{a}$ 、 $\boldsymbol{b}$ 有什么样的位置关系（在 3.4 节中有用到）。二维向量的情况下，其位置关系如图 6-5-4 所示。在游戏中，基于多边形渲染的自由形状的多角形物体进行碰撞检测时，除了检测物体是否碰撞外，如果还想知道多边形的哪个边没有碰撞，使用上面的公式会比较方便。

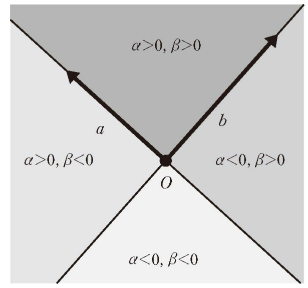


图 **6-5-4**　二维向量中向量间位置关系的判定

- 旋转矩阵

  关于向量上文中已经做了一些介绍，向量可以通过矩阵 对其进行线性变换 。在数学上矩阵的应用范围非常广泛，本小节仅对二维向量进行线性变换时所用到的一些矩阵（都是 2 行 2 列的矩阵）进行说明。

  可用于二维向量线性变换的矩阵，是 2 行 2 列的正方形矩阵，形如

  $$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

  将其作用于一个二维向量 $\begin{pmatrix} v_x \\ v_y \end{pmatrix}$ （这样写表示向量是一个列向量）时需要如下操

作。

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

即将矩阵放于列向量左侧并相乘，相乘的结果与变换前相同，仍然是一个列向量，具体为

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} av_x + bv_y \\ cv_x + dv_y \end{pmatrix}$$

不过看完是不是仍然有点摸不着头脑呢？这样的变换到底有什么作用，似乎之前没有见过对应的实例。其实在线性变换的矩阵中，有一个是游戏开发必不可少的，那就是旋转矩阵 ，它可以将向量旋转任意角度。通过旋转矩阵，无论是怎样的长度和初始方向的向量，都可以将其旋转到所需要的角度，仅从这点就可以知道矩阵是游戏开发中必不可少的工具了。

那么旋转矩阵具体是怎样的呢？将向量旋转 θ 的旋转矩阵，只需要将 x 方向的基向

量 $\begin{pmatrix}1\\0\end{pmatrix}$ 变换为 $\begin{pmatrix}\cos\theta\\\sin\theta\end{pmatrix}$，$y$ 方向的基向量 $\begin{pmatrix}0\\1\end{pmatrix}$ 变换为 $\begin{pmatrix}-\sin\theta\\\cos\theta\end{pmatrix}$ 即可（参考图 6-5-5）。

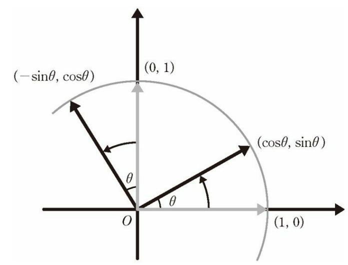


图 **6-5-5**　向量的旋转

那么令旋转矩阵为 $\begin{pmatrix}a & b\\c & d\end{pmatrix}$，将 $\begin{pmatrix}1\\0\end{pmatrix}$ 变换为 $\begin{pmatrix}\cos\theta\\\sin\theta\end{pmatrix}$，则有

$$\begin{pmatrix}a & b\\c & d\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix}=\begin{pmatrix}\cos\theta\\\sin\theta\end{pmatrix}$$

左边进行乘法运算，可得

$$\begin{pmatrix}a\\c\end{pmatrix}=\begin{pmatrix}\cos\theta\\\sin\theta\end{pmatrix}$$

$\therefore a=\cos\theta,\ c=\sin\theta$

同时，将 $\begin{pmatrix}0\\1\end{pmatrix}$ 变换为

$\begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix}$，则有

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -sin\theta \\ \cos\theta \end{pmatrix}$$

左边进行乘法运算，有

$$\begin{pmatrix} b \\ d \end{pmatrix} = \begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix}$$

$$\therefore b = -\sin\theta,\ \ d = \cos\theta$$

至此 a 、b 、c 、d 就已经全部求出来了，代入原来的矩阵 $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$，可得

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

也就是说，将向量 $\begin{pmatrix} v_x \\ v_y \end{pmatrix}$ 旋转角度 θ，只需将上面的旋转矩阵放在向量左边并相乘，然后进行计算即可，如下所示。

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} v_x\cos\theta - v_y\sin\theta \\ v_x\sin\theta + v_y\cos\theta \end{pmatrix}$$

通过这种方式，无论向量 $\begin{pmatrix} v_x \\ v_y \end{pmatrix}$ 的初始长

度和方向如何，都可以将其旋转 θ ，非常方便。想要从事游戏开发的朋友，请务必牢记这个重要的旋转矩阵。

- 单位矩阵 $\boldsymbol{E}$

  除了旋转矩阵，在游戏开发中还有一些重要的矩阵，其中之一就是单位矩阵 $\boldsymbol{E}$ 。所谓单位矩阵，就是一个向量与其相乘后，仍然等于原来的向量的矩阵。在普通的数学计算中，一个数乘以 1 仍然等于原来的数，而单位矩阵就相当于数字 1。具体来说，单位矩阵是一个 2 行 2 列的矩阵，形如

  $$E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

  有兴趣的读者可以试试将其乘以向量 $\begin{pmatrix} v_x \\ v_y \end{pmatrix}$ ，看等式

  $$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

  是否真的成立，如果成立，就可以确认 E

乘以一个向量后向量仍然保持不变。

- 逆矩阵

  最后不能不提的是逆矩阵 这一特殊的矩阵。逆矩阵对应的是逆变换。与普通矩阵所进行的变换相反，逆变换可以将矩阵已经作用的变换取消，而实现这种逆变换的矩阵就称为逆矩阵。单看定义可能有些复杂，比如有将向量顺时针旋转 θ 的旋转矩阵，与之相对的逆矩阵就是将向量逆时针旋转 θ，因为顺时针旋转后又逆时针转回，等于没有旋转，这样就等于取消了原矩阵作用的变换。

  在游戏开发中，要用到逆变换的场景非常多，因此逆矩阵也非常重要。其实游戏开发中通常用到的一些变换，包括 3D 变换，都存在对应的逆变换，因此也就有相应的逆矩阵存在。具体来说，以 2D 中 2 行 2 列的矩阵为例，假设

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

其逆矩阵为

$$A^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

其中 A^{-1} 的上标 -1 就表示这是一个逆矩阵， $\boldsymbol{A}^{-1}$ 读作 ***A*** 逆。

下面就让我们实际使用这个公式，来试验一下旋转是否能变成逆时针旋转吧。已知将 2D 向量旋转角度 θ 的矩阵为

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

将其对应于上式的矩阵 A，则有 $a=\cos\theta$、$b=-\sin\theta$、$c=\sin\theta$、$d=\cos\theta$，那么其逆矩阵 A^{-1} 为

$$\frac{1}{(\cos\theta)^2 + (\sin\theta)^2} \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

由于其中 $\cos\theta$ 与 $\sin\theta$ 分别代表半径为 1 的单位圆上，角度 θ 所对应的 x 坐标与 y 坐标，根据勾股定理可知 $(\cos\theta)^2 + (\sin\theta)^2 = 1^2 = 1$，代入上

面的逆矩阵有

$$\frac{1}{1}\begin{pmatrix}\cos\theta & \sin\theta \\ -\sin\theta & \cos\theta\end{pmatrix} = \begin{pmatrix}\cos\theta & \sin\theta \\ -\sin\theta & \cos\theta\end{pmatrix}$$

又由于 cos (-θ) = cos θ ，并且 sin (-θ) = -sin θ ，于是有

$$\begin{pmatrix}\cos\theta & \sin\theta \\ -\sin\theta & \cos\theta\end{pmatrix} = \begin{pmatrix}\cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta)\end{pmatrix}$$

最后得到的矩阵，就是将原旋转矩阵的 θ 替换为了 -θ ，证明这个逆矩阵确实是对旋转角度 θ 的逆旋转。

将原矩阵与逆矩阵相乘，会得到上一小节中出现过的单位矩阵 E ，具体过程这里不再详细介绍。原理应该不难理解，一个向量乘以单位矩阵后不会发生改变，而逆矩阵正好可以取消矩阵产生的变换，因此逆矩阵与矩阵相乘的效果与单位矩阵完全一致。读者可以将矩阵与逆矩阵的关系，类比为普通数学计算中一个数与其倒数的关系，因为一个数乘以其倒数也正好等于 1。

6.6　微分

Key Word　变化率、微分系数、极限、合成函数

- 微分

  所谓微分，可以看作是一种求微分系数的操作。而微分系数，是指某个函数关于某个值的变化率。这个概念在游戏开发中经常被用来考察某个函数变化的速度。举个具体的例子，比如想要知道一条直线 $f(x) = ax$ 的变化率，首先令 x 只变化 Δx 时 $f(x)$ 的变化率为 d，则有

$$\begin{aligned} d &= \frac{f(x)\text{的变化量}}{x\text{的变化量}} \\ &= \frac{f(x+\Delta x)-f(x)}{\Delta x} \\ &= \frac{a(x+\Delta x)-ax}{\Delta x} \\ &= \frac{a\Delta x}{\Delta x} \\ &= a \end{aligned}$$

因此 $f(x) = ax$ 的变化率，也就是微分常数总是为 a 。对于一条直线，一眼看去就能明白，其斜率就是直线函数的变化率。

接下来考虑 $f(x) = ax + b$ 的微分系数，当 x 仅变化 Δx 时，$f(x)$ 的变化率为

$$\begin{aligned} d &= \frac{f(x+\Delta x)-f(x)}{\Delta x} \\ &= \frac{\{a(x+\Delta x)+b\}-(ax+b)}{\Delta x} \\ &= \frac{a\Delta x}{\Delta x} \\ &= a \end{aligned}$$

虽然增加了常数 b ，但是其变化率，即微分系数仍然为 a 。这可以理解为虽然出发地点不同，但只要速度不变，位置的变化率也应该相同。说明函数即使加上一个常数，其微分系数也不会变化。

我们已经知道表

示直线的函数 $f(x) = ax + b$ 的微分系数始终为常数 a。那么非直线，即表示曲线的函数，像抛物线 $f(x) = ax^2 + b$ 的情况又如何呢？在抛物线上，变化的速度根据位置的不同而有所区别，因此大致可以猜想到，$f(x)$ 的变化率应该是一个关于 x 的函数。让我们实际计算一下变化率 d 吧。

$$\begin{aligned} d &= \frac{f(x+\Delta x) - f(x)}{\Delta x} \\ &= \frac{a(x+\Delta x)^2 - ax^2}{\Delta x} \\ &= \frac{ax^2 + 2ax(\Delta x) + a(\Delta x)^2 - ax^2}{\Delta x} \\ &= 2ax + a(\Delta x) \end{aligned}$$

可以看到 $f(x)$ 的变化率会同时受到 x 与 Δx 双方的影响。对于抛物线 $f(x)= ax^2$，x 改变时 变化率会改变（图 6-6-1 左），同时，当 x 的变化量 Δx 改变时，变化率也会改变（图 6-6-1 右）。

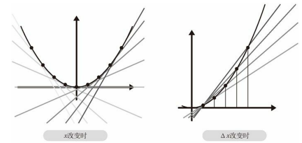


图 **6-6-1**　抛物线 $\boldsymbol{f(x) = ax^2}$ 的变化率

我们已经知道微分系数的定义是 $f(x)$ 关于 x 的变化率。对于一个特定的 x，总会得到一个对应的变化率。比如在物体运动的某一个瞬间，必定有一个对应该时刻的速度。但是变化率的本质是 $\frac{f(x)\text{变化量}}{x\text{的变化量}}$，如果我们只考虑某个特定的 x，当 x 的变化量减小为 0 时，变化率也就无从得知了。为了维持 x 的变化量始终有值，且让每个 x 都能有自己的微分系数存在，我们需要执行一个特殊操作，就是让 x 的变化量 Δx 无限小。这个操作表示为记号可以写作 $\lim_{\Delta x \to 0} \bigcirc$，称

为极限 。对变化率取极限得到的微分系数写作 $f'(x)$，而通过这种方式求微分系数的过程就叫作微分 。下面用数学符号来表示对抛物线 $f(x) = ax^2$ 进行微分的过程。

$$\begin{aligned} f'(x) &= \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \\ &= \lim_{\Delta x \to 0} (2ax + a(\Delta x)) \\ &= 2ax \end{aligned}$$

对于一个抛物线来说，x 越大 x 的变化率也越大。而要求某个点的变化率时，可以画出通过该点并与曲线相切的直线，这条直线的斜率就是该点的变化率，可以很明显地看出，x 越大切线的斜率也越大（参考图 6-6-1）。

表示微分系数的符号，除了 $f'(x)$ 外，还可以写成 $\frac{\mathrm{d}}{\mathrm{d}x}f(x)$ ，另外，当 $y = f(x)$

时，还可以写作 $\frac{\mathrm{d}y}{\mathrm{d}x}$。这种写法可以很明确地表示正在对哪一个变量进行微分。比如 $\frac{\mathrm{d}y}{\mathrm{d}x}$ 就表示对 y 进行关于 x 的微分。

在上面的例子中，我们分别对直线（x 的一次方）、抛物线（x 的二次方）求了微分系数。那么对于更高次数（如 x 的三次方、x 的四次方）该如何处理呢？在游戏开发中，比如当需要在任意坐标放置多个点并作出通过这些点的补间曲线 时，有时就会用到这种高次数的微分。接下来为了更好地理解，我们分别对 x 的一次方、二次方、三次方等进行微分。

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}x}(x) &= \lim_{\Delta x \to 0} \frac{(x + \Delta x) - x}{\Delta x} \\
&= \lim_{\Delta x \to 0} \frac{\Delta x}{\Delta x} \\
&= 1 \\
\frac{\mathrm{d}}{\mathrm{d}x}(x^2) &= \lim_{\Delta x \to 0} \frac{(x + \Delta x)^2 - x^2}{\Delta x} \\
&= \lim_{\Delta x \to 0} \frac{x^2 + 2x(\Delta x) + (\Delta x)^2 - x^2}{\Delta x} \\
&= \lim_{\Delta x \to 0} \left\{ \frac{2x(\Delta x)}{\Delta x} + \frac{(\Delta x)^2}{\Delta x} \right\} \\
&= \lim_{\Delta x \to 0} (2x + \Delta x) \\
&= 2x \\
\frac{\mathrm{d}}{\mathrm{d}x}(x^3) &= \lim_{\Delta x \to 0} \frac{(x + \Delta x)^3 - x^3}{\Delta x} \\
&= \lim_{\Delta x \to 0} \frac{x^3 + 3x^2(\Delta x) + 3x(\Delta x)^2 + (\Delta x)^3 - x^3}{\Delta x} \\
&= \lim_{\Delta x \to 0} \left\{ \frac{3x^2(\Delta x)}{\Delta x} + \frac{3x(\Delta x)^2}{\Delta x} + \frac{(\Delta x)^3}{\Delta x} \right\} \\
&= \lim_{\Delta x \to 0} \{3x^2 + 3x\Delta x + (\Delta x)^2\} \\
&= 3x^2
\end{aligned}
$$

…（以下省略）…

这里不再一一列举更高次方的情况，一般来说微分可以表示为

$$\frac{\mathrm{d}}{\mathrm{d}x}(x^n) = nx^{n-1}$$

下面列举一些与微分相关的各种计算的公式。

- 加法 · 减法

一般来说

$$\frac{\mathrm{d}}{\mathrm{d}x}(f(x) + g(x)) = \frac{\mathrm{d}}{\mathrm{d}x}f(x) + \frac{\mathrm{d}}{\mathrm{d}x}g(x)$$

即将函数相加后进行微分，等于将函数分别微分后相加。比如

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}x}(x^2 + x) &= \frac{\mathrm{d}}{\mathrm{d}x}(x^2) + \frac{\mathrm{d}}{\mathrm{d}x}(x) \\
&= 2x + 1
\end{aligned}
$$

同理，对于减法

有

$$\frac{\mathrm{d}}{\mathrm{d}x}(f(x)-g(x))=\frac{\mathrm{d}}{\mathrm{d}x}f(x)-\frac{\mathrm{d}}{\mathrm{d}x}g(x)$$

- 常数倍

  一般来说，对于常数 a 有

  $$\frac{\mathrm{d}}{\mathrm{d}x}(af(x))=a\frac{\mathrm{d}}{\mathrm{d}x}f(x)$$

  即对函数的 a 倍进行微分，等于微分后的函数乘以 a 倍。这可以再与加法运算组合，比如

  $$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}x}(ax^3+bx^2+cx)&=\frac{\mathrm{d}}{\mathrm{d}x}(ax^3)+\frac{\mathrm{d}}{\mathrm{d}x}(bx^2)+\frac{\mathrm{d}}{\mathrm{d}x}(cx)\\&=a\frac{\mathrm{d}}{\mathrm{d}x}(x^3)+b\frac{\mathrm{d}}{\mathrm{d}x}(x^2)+c\frac{\mathrm{d}}{\mathrm{d}x}(x)\\&=3ax^2+2bx+c\end{aligned}$$

  对于如三角函数这样不含多次方项的函数，考虑其微分的情况，有

  $$\frac{\mathrm{d}}{\mathrm{d}x}(\sin x)=\lim_{\Delta x\to 0}\frac{\sin(x+\Delta x)-\sin x}{\Delta x}$$

  然后运用三角函数的积化和差公式

  $$\sin A-\sin B=2\cos\frac{A+B}{2}\sin\frac{A-B}{2}$$

  可得

  $$\frac{\mathrm{d}}{\mathrm{d}x}(\sin x)=\lim_{\Delta x\to 0}\frac{2\cos\frac{2x+\Delta x}{2}\sin\frac{\Delta x}{2}}{\Delta x}$$

将 lim 部分的分数的分子分母同除以 2，有

$$\frac{\mathrm{d}}{\mathrm{d}x}(\sin x)=\lim_{\Delta x\to 0}\left(\cos\frac{2x+\Delta x}{2}\cdot\frac{\sin\frac{\Delta x}{2}}{\frac{\Delta x}{2}}\right)$$

对于前面的 cos 部分，有

$$\lim_{\Delta x\to 0}\left(\cos\frac{2x+\Delta x}{2}\right)=\cos\frac{2x}{2}=\cos x$$

对于后面的 sin 部分，因为有公式

$$\lim_{t\to 0}\left(\frac{\sin t}{t}\right)=1$$

（这个公式可以使用逼近法进行证明，此处省略），令 $\frac{\Delta x}{2}=t$，则有

$$\lim_{\Delta x\to 0}\left(\frac{\sin\frac{\Delta x}{2}}{\frac{\Delta x}{2}}\right)=\lim_{t\to 0}\left(\frac{\sin t}{t}\right)=1$$

最后可得

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}x}(\sin x)&=\lim_{\Delta x\to 0}\left(\cos\frac{2x+\Delta x}{2}\cdot\frac{\sin\frac{\Delta x}{2}}{\frac{\Delta x}{2}}\right)\\&=\cos x\end{aligned}$$

这样就完成了对 sin*x* 的微分。对于 cos 可以使用同样的方法，最终得到

$$\frac{\mathrm{d}}{\mathrm{d}x}(\cos x) = -\sin x$$

将上面得到的微分公式整理如下。

$$\frac{\mathrm{d}}{\mathrm{d}x}(x) = 1$$
$$\frac{\mathrm{d}}{\mathrm{d}x}(x^2) = 2x$$
$$\frac{\mathrm{d}}{\mathrm{d}x}(x^3) = 3x^2$$
$$\frac{\mathrm{d}}{\mathrm{d}x}(x^n) = nx^{n-1}$$
$$\frac{\mathrm{d}}{\mathrm{d}x}(\sin x) = \cos x$$
$$\frac{\mathrm{d}}{\mathrm{d}x}(\cos x) = -\sin x$$

除了上面的公式外，还需要知道如果微分不含变量，即仅对常数进行微分的话会得到 0。这些公式基本上覆盖了游戏开发中所涉及的有关微分的知识要点，希望读者可以记住这些结论。

- 合成函数的微分

  最后介绍一点关于合成函数 的微分知识。在游戏开发中可能会

遇到 $\sin(\omega t)$ 或 $\cos(\omega t)$ 这样的等式（ω 为常数，t 为时间）并需要对 t 进行微分的情况，此时与 t 相乘的常数 ω 就显得比较碍事，因为它导致三角函数的微分公式无法直接使用。当然可以改用求极限的 lim 等式，但是实际操作起来需要一步步计算，仍然很麻烦，所以我们还是希望能直接使用已有的公式。值得庆幸的是，我们还有合成函数的微分这一便利的工具，接下来就使用它对 $\sin(\omega t)$ 微分试试吧。首先令 $y=\sin(\omega t)$、$x=\omega t$，可得

$$\begin{cases} y = \sin x \\ x = \omega t \end{cases}$$

这样一来只要求得 $\frac{\mathrm{d}y}{\mathrm{d}t}$，就等于求出了

$\frac{\mathrm{d}}{\mathrm{d}t}(\sin(\omega t))$。但是上面的等式中 y 被表示为了关于 x 的函数，不能直接对 y 求 t 的积分。而此时就轮到合成函数的微分出场了。

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \frac{\mathrm{d}y}{\mathrm{d}x} \cdot \frac{\mathrm{d}x}{\mathrm{d}t}$$

这个等式如果仅仅作为分数的话当然是成立的，但要注意它不是分数而是微分。因此这个等式的含义是：对 y 求 t 的微分所得的结果，等于对 y 求 x 的微分的结果乘以对 x 求 t 的微分的结果。实际来试试看吧。

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}x}(\sin x) = \cos x$$
$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}(\omega t) = \omega$$

因此有

$$\frac{\mathrm{d}}{\mathrm{d}t}(\sin(\omega t)) = \frac{\mathrm{d}y}{\mathrm{d}t} = \frac{\mathrm{d}y}{\mathrm{d}x} \cdot \frac{\mathrm{d}x}{\mathrm{d}t} = \cos x \cdot \omega = \omega \cos x$$

再将 $x = \omega t$ 代入 cos，最后得到

$$\frac{\mathrm{d}}{\mathrm{d}t}(\sin(\omega t)) = \omega\cos(\omega t)$$

同理，还可以得到

$$\frac{\mathrm{d}}{\mathrm{d}t}(\cos(\omega t)) = -\omega\sin(\omega t)$$

上面的计算在 1.6 节中有使用。通过上面的说明，其实可以将合成函数的微分不太严谨地看作“对函数中又包含另一层函数的等式进行微分，可以将其转化为对外层函数的微分乘以对内层函数的微分”，这样可能比较方便记忆。

下面再举一个应用合成函数的微分的例子，对等式

$$y = (at+b)^2$$

如何进行微分呢？此时先将等式展开得到

$$y = a^2t^2 + 2abt + b^2$$

虽然计算

$$\frac{\mathrm{d}y}{\mathrm{d}t} = 2a^2t + 2ab$$

也可以求解，但是使用合成函数的微分会更加简单，令

$$x = at + b$$

则有

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \frac{\mathrm{d}y}{\mathrm{d}x} \cdot \frac{\mathrm{d}x}{\mathrm{d}t} = 2(at + b) \cdot a = 2a^2t + 2ab$$

展开后与采用普通微分得到的结果一样。这种方法可以回避等式中复杂的展开，免去不少麻烦，希望读者朋友善加使用。

6.7 级数与积分

Key Word 数列、西格玛、原函数、不定积分、积分常数

- 级数

所谓级数，就是将一个数列的每一项加起来得到的结果。而这里所说的数列，是按照一定规则排列的一系列数字。比如下面就是一个数列。

$a_1 = 1$，a_2 =

2，a_3 = 3，a_4 = 4，a_5 = 5，……

而下面同样也是一个数列。

a_1 = 1，a_2 = 2，a_3 = 4，a_4 = 8，a_5 = 16，……

对于开发人员来说，将数列对应为数组变量可能会比较容易理解。但是与程序中的数组变量不同，数学中的数列大多都可以将内容（数字）以算式的形式表示。比如上面的

a_1 = 1，a_2 = 2，a_3 = 3，a_4 = 4，a_5 = 5，

……

这一数列可以表示为

$a_n = n$

而

$a_1 = 1$，$a_2 = 2$，$a_3 = 4$，$a_4 = 8$，$a_5 = 16$，……

这一数列则可以表示为

$a_n = 2^{n-1}$

由于级数是将某个数列的一个编号到另一个编号之间的数字全部相加，因此级数也可以用符号表示为

$$\sum_{i=m}^{n} a_i$$

这表示将数列的第 m 个数字到第 n 个数字全

部相加所得到的数。读作“西格玛 a 从 m 到 n ”。举个简单的例子。

$$\sum_{i=1}^{5} 1$$

这种情况下，由于 ∑ 内只有常数 1，不包含每次变化的 i ，因此表示 1+1+1+1+1+ 即 1 相加 5 次。所以

$$\sum_{i=1}^{5} 1 = 5$$

下面举一个稍微复杂的例子。

$$\sum_{i=1}^{5} i$$

此时 $a_i = i$ ，也就是说 $a_1 = 1$, $a_2 = 2$ $a_3 = 3$, $a_4 = 4$ $a_5 = 5$,

……，然后将第 1 个到第 5 个数字全部加起来，即将从 1 到 5 的数全部相加，1+2+3+4+5=15，所以

$$\sum_{i=1}^{5} i = 15$$

那么如果上限不为 5，而是做从 1 到 *n* 的加法，即

$$\sum_{i=1}^{n} i$$

的值为多少呢？虽然我们很容易能知道 *n* =1 时为 1，*n* =2 时 1+2=3，*n* =3 时 1+2+3=6……但是一直计算到 *n* 是不是有点难呢。其实对于 1～*n* 的加法，只要

增加一个倒序的数列就可以很容易地求解。假设

$$x = \sum_{i=1}^{n} i$$

有

$$\begin{aligned} x &= 1 + 2 + 3 + \cdots + (n-1) + n \\ x &= n + (n-1) + (n-2) + \cdots + 2 + 1 \end{aligned}$$

然后将上下两个式子相对应的项相加，有

$2x = (n + 1) + (n + 1) + (n + 1) + \ldots +(n + 1) + (n + 1)$

对原数列加上一个倒序的数列后，就变成相同数字的加法了。由于相加的数一共有 n 个，因此右边有 n 个 $(n +1)$，即

$$2x = n(n+1)$$
$$\therefore x = \frac{1}{2}n(n+1)$$

所以

$$\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$$

下面将 *n* =5 代入试试。

$$\sum_{i=1}^{5} i = \frac{1}{2} \cdot 5 \cdot (5+1) = 15$$

与之前的计算结果一致。这种对 1～*n* 的级数求解的方法，在计算其他级数时一般也能适用，最好可以记住。

- 不从 **1** 开始的级数

接下来让我们思考如果数列不从 1 开始会怎样。比如

$$\sum_{i=m}^{n} i$$

这种情况，表示从 *m* 到 *n* 的整数全部相加，比如当 *m*

=3、*n* =7 时，有 3+4+5+6+7=那么级数用 *m* 与 *n* 表示时要如何计算呢？与之前的从 1 开始的级数相同，我们对其加上一个逆序的级数。假设

$$x = \sum_{i=m}^{n} i$$

则有

$$x = m + (m+1) + (m+2) + \cdots + (n-1) + n$$
$$x = n + (n-1) + (n-2) + \cdots + (m+1) + m$$

然后将上下两式相加，有

2*x* = (*m* + *n*) + (*m* + *n*) + (*m* + *n*) + … +(*m* + *n*) + (*m* + *n*)

相加的数从 *m* ～*n* 一共有 (*n* -*m* +1) 个。比如若 *m* =3、*n* =7，则一共有 (7-

3+1)=5个数字。即等式右边共有(*n* -*m* +1)个(*m* +*n*)，因此

$$2x=(m+n)(n-m+1)$$
$$\therefore x=\frac{1}{2}(m+n)(n-m+1)$$

最终得到

$$\sum_{i=m}^{n} i=\frac{1}{2}(m+n)(n-m+1)$$

让我们代入*m* =3、*n* =7验证一下。

$$\sum_{i=3}^{7} i=\frac{1}{2}(3+7)(7-3+1)=\frac{1}{2}\cdot 10\cdot 5=25$$

与之前计算的结果一致。有兴趣的读者可以自行验证一下*m* =1时结果与之前*i* 从1开始的数列是否一致。

- 复杂算式的级数

最后介绍一下∑内的算式比较复杂时，将其分割为多个

$\sum$ 以便使用公式的方法。比如

$$\sum_{i=m}^{n}(a \cdot i + b)$$

（a , b 为常数）

这个算式，可以将其分割为

$$\sum_{i=m}^{n}(a \cdot i + b) = a\sum_{i=m}^{n} i + b\sum_{i=m}^{n} 1$$

像这样，将常数移到 $\sum$ 的外面，就可以分割为多个 $\sum$ 了。如果觉得不好理解，可以类比一下加法中乘以共同的系数时，可以先与系数相乘再相加，同时更改加法的顺序也不会影响结果（在 1.4 节中使用过）。

- 积分

上面已经介

绍了级数相关的知识。积分 与级数有相似的部分，从形式上来看是微分的逆运算，比如在求图形的面积，或者由加速度计算速度、由速度计算位置等操作时会使用到。对位置关于时间微分就会得到速度，而当速度已知要求位置时，我们其实是在计算被微分前原来的函数是什么，也就是在求原函数 。求原函数的操作就是积分。比如

$$\frac{\mathrm{d}}{\mathrm{d}x}(x^2) = 2x$$

那么 $2x$ 的积分是不是就等于 x^2 呢？事实上

微分后等于 2*x* 的函数并不只有 1 个。因为在微分过程中会将常数消去，假设 *C* 为常数，那么

$$\frac{\mathrm{d}}{\mathrm{d}x}(x^2+C)=2x$$

2*x* 的积分等于 x^2+C 才是正确的。积分可以用以下符号表示。

$$\int 2x\mathrm{d}x=x^2+C$$

像这样结果中含有不确定常数（上式中为 *C* ）的积分称为不定积分，*C* 称为积分常数 。还是以速度与位置的关系为例，令时间为 *t* ，那么以 2*t* 的速度（即随时间加速）运动的

物体，到达的位置就为 t^2+C。当 $t=0$ 时位置为 C，这里的积分常数 C 就代表了物体的初始位置。同样，通过对加速度积分来求速度时，积分常数就表示初速度（参考 1.4 节及 1.7 节）。

接下来列举一些积分相关的公式。首先根据对 x^n 进行微分的一般形式可得

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{x^{n+1}}{n+1}\right)=x^n$$

对等式两边进行积分，有

$$\int x^n\mathrm{d}x=\left(\frac{x^{n+1}}{n+1}\right)+C$$

具体如

$$
\begin{aligned}
\int x\mathrm{d}x &= \frac{x^2}{2} + C \\
\int x^2\mathrm{d}x &= \frac{x^3}{3} + C \\
&\vdots
\end{aligned}
$$

依次类推即可。

而与微分相同，积分的情况下也有以下公式成立。

- 加法·减法

  一般有

  $$\int (f(x) + g(x))\mathrm{d}x = \int f(x)\mathrm{d}x + \int g(x)\mathrm{d}x$$

  即函数相加后进行积分，等于函数分别积分后再相加。比如

  $$
  \begin{aligned}
  \int (x^2 + x + 1)\mathrm{d}x &= \int x^2\mathrm{d}x + \int x\mathrm{d}x + \int x^0\mathrm{d}x \\
  &= \frac{x^3}{3} + \frac{x^2}{2} + x + C
  \end{aligned}
  $$

  （请注意 $x^0 = 1$）

  注意结果中将

三部分得到的积分常数整合为一个了。如果对此理解有困难的话，可以将得到的结果进行微分看能否还原回去。

同理，对于减法有

$$\int (f(x) - g(x))\mathrm{d}x = \int f(x)\mathrm{d}x - \int g(x)\mathrm{d}x$$

- 常数倍

  一般来说对于常数 a 有

  $$\int (af(x))\mathrm{d}x = a\int f(x)\mathrm{d}x$$

  即对函数的 a 倍进行积分，等于对函数积

分后乘以 a 倍。将其与加法运算组合，则有

$$\begin{aligned}\int(ax^2+bx+c)\mathrm{d}x &= \int(ax^2)\mathrm{d}x+\int(bx)\mathrm{d}x+\int(cx^0)\mathrm{d}x \\ &= a\int x^2\mathrm{d}x+b\int x\mathrm{d}x+c\int x^0\mathrm{d}x \\ &= \frac{ax^3}{3}+\frac{bx^2}{2}+cx+C\end{aligned}$$

与微分不同的是，积分并不是一定可以进行的。也就是说，对一个函数进行积分时，并不能保证一定可以得到一个已知的函数。因此在用计算机进行积分计算时，一般都不是为了进行非常严

密的计算，大多都是在允许一定程度的误差的基础上通过数值计算求解，特别是在游戏开发中尤甚，几乎完全依赖数值计算。读者朋友如果遇到根据速度求位置等必须要用到积分的情况，应当首先考虑能否容忍数值计算的误差，是否真的需要严密的求

解等，
然后再
根据实
际情况
选择解
决方
法。

附录 示例程序的编译及运行方法——基于 **Visual Studio 2013、**

Visual Studio 2012、Visual Studio 2010

本小节将介绍基于 Visual Studio 编译及运行本书示例代码的方法。本书的示例程序中，除了数学、物理学的理论知识的实践之外，还用到了 DirectX 11。因此，为了编译本书的示例代码，需要安装 Visual Studio 以及 DirectX 11 SDK（Softw Developmen

Kit）。请注意我们需要的是SDK版本，而不是一般玩游戏时预装的Runtime版本，仅安装Runtime版本将无法完成编译。

DirectX 11 SDK可从以下网址下载，到目前为止（2014年9月），最新版本为9.29.1962，文件大小为571.7 MB。

- Microso
  Downlo
  Center

  ⇒
  http://w
  us/down
  id=6812

POINT
如果只是运行示例程序的

可执行文件（exe文件）的话，仅安装DirectX 11的Runtime版本就可以了。

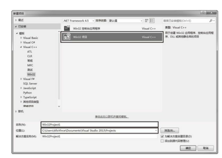
操作步骤

首先需要新建项目，类型是基于C++的Win32项目，具体操作方法如下。

1. 在Visual Studio的“文件”菜单中选择“新建”→“项目”。

2. 在“新建项目”对话框中，在左侧分类中找到“已安装”→“模

板”→“Visua C++”→“Wir 选中后在中间的列表中选择“Win32项目”，同时可以在下方的“名称”中填入项目名称，在“位置”中填入项目代码的保存位置，最后点击“确认”按钮。



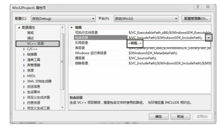
3. 在弹出的“Win32应用程序向导”对话框中，点击左侧的“应用程序设置”，然后勾选“附加项目”下的“空项目”，最后点击“完成”按钮。



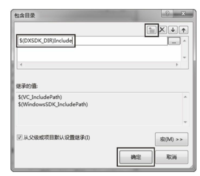
这样就创建出了项目文件夹。接下来就需要让项目识别DirectX的包含目录及库目录，具体操作如下。

4. 在Visual Studio的“项目”菜单中选择“属性”。

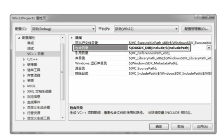
5. 在弹出的项目“属性页”对话框中，点击左侧的“配置属性”→“VC++目录”，选中“包含目录”，这时在最右端会出现▼，点击该图标后会出现“编辑”选项，点击“编辑”。



6. 在弹出的“包含目录”对话框中，点击第一行第一个图标会出现输入框，在输入框中输入$(DXSDK_I点击“确定”按钮。

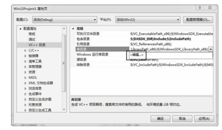


7. 返回“属性页”对话框，可以看到“包含目录”这一栏已经显示为了“$(DXSDI



8. 接下来，点击中间的“库目录”，同样点击▼，再

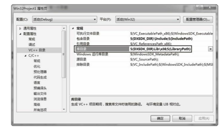
点击随后出现的“编辑”选项。



9. 在弹出的“库目录”对话框中，点击第一行第一个图标会出现输入框，在输入框中输入$(DXSDK_I
点击“确定”按钮。



10. 返回“属性页”对话框，可以看到“库目录”这一栏已经显示为了“$(DXSDI
确认后点击“确定”按钮。



这样一来，项目就可以识别DirectX所必需的一些目录了。接下来需要向项目中添加代码文件并编译，具体操作如下。

11. 找到项目所在文件夹（比如Test项目所对应的就是Test.vcxproj文件所在的文件夹），将想要编译的源代码文件（如Movement_1等）、着色器文件（Basic_2D.图片素材等必需的文件复制进去。

12. 在Visual Studio中找

到“解决方案资源管理器”（如果没有找到可以通过菜单的“视图”→“解决方案资源管理器”开启），在“源文件”选项上点击右键，选择“添加”→“现有项”，然后选择想要编译的源代码（注意，此处只需选择 cpp 代码即可，无需添加 .fx 文件），并点击“添加”按钮。

13. 在“解决方案资源管理器”的“源文件”中，能看到想要编译的源代码文件被添加了进来，双击该文件就可以打

开。



这样就将源代码添加到了项目中。接下来就可以进行编译和运行了。

14. 点击 Visual Studio 上方的本地 Windows 调试器。



15. 这时 Visual Studio 会开始编译，编译成功后将自动运行示例程序。

-
    -
        -
            -
                -
                -

看完了

如果您对本书内容有疑问，可发邮件至contact@turi会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turin

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_
讲述码农精彩人生
- 微信 图灵教育：turingbc
